

Quick Reference

lisp cl

Common

lisp

Bert Burgemeister

Common Lisp Quick Reference Revision 145 [2018-01-15]
Copyright © 2008–2018 Bert Burgemeister
L^AT_EX source: <http://clqr.boundp.org>



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

Contents

1	Numbers	3	9.5	Control Flow	19
1.1	Predicates	3	9.6	Iteration	20
1.2	Numeric Functions	3	9.7	Loop Facility	21
1.3	Logic Functions	4	10	CLOS	23
1.4	Integer Functions	5	10.1	Classes	23
1.5	Implementation-Dependent	6	10.2	Generic Functions	25
2	Characters	6	10.3	Method Combination Types	26
3	Strings	7	11	Conditions and Errors	27
4	Conses	8	12	Types and Classes	29
4.1	Predicates	8	13	Input/Output	31
4.2	Lists	8	13.1	Predicates	31
4.3	Association Lists	9	13.2	Reader	32
4.4	Trees	10	13.3	Character Syntax	33
4.5	Sets	10	13.4	Printer	34
5	Arrays	10	13.5	Format	36
5.1	Predicates	10	13.6	Streams	38
5.2	Array Functions	10	13.7	Pathnames and Files	40
5.3	Vector Functions	11	14	Packages and Symbols	41
6	Sequences	12	14.1	Predicates	41
6.1	Sequence Predicates	12	14.2	Packages	41
6.2	Sequence Functions	12	14.3	Symbols	43
7	Hash Tables	14	14.4	Standard Packages	44
8	Structures	15	15	Compiler	44
9	Control Structure	15	15.1	Predicates	44
9.1	Predicates	15	15.2	Compilation	44
9.2	Variables	16	15.3	REPL and Debugging	45
9.3	Functions	17	15.4	Declarations	46
9.4	Macros	18	16	External Environment	46

Typographic Conventions

name ; <i>f</i> name ; <i>g</i> name ; <i>m</i> name ; <i>s</i> name ; <i>v</i> * name *; <i>c</i> name	▷ Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.
<i>them</i>	▷ Placeholder for actual code.
me	▷ Literal text.
[<i>foo</i> <u><i>bar</i></u>]	▷ Either one <i>foo</i> or nothing; defaults to bar .
<i>foo</i> *; { <i>foo</i> }*	▷ Zero or more <i>foos</i> .
<i>foo</i> ⁺ ; { <i>foo</i> } ⁺	▷ One or more <i>foos</i> .
<i>foos</i>	▷ English plural denotes a list argument.
{ <i>foo</i> <i>bar</i> <i>baz</i> }; $\begin{cases} \textit{foo} \\ \textit{bar} \\ \textit{baz} \end{cases}$	▷ Either <i>foo</i> , or <i>bar</i> , or <i>baz</i> .
$\begin{cases} \textit{foo} \\ \textit{bar} \\ \textit{baz} \end{cases}$	▷ Anything from none to each of <i>foo</i> , <i>bar</i> , and <i>baz</i> .
$\widehat{\textit{foo}}$	▷ Argument <i>foo</i> is not evaluated.
$\widetilde{\textit{bar}}$	▷ Argument <i>bar</i> is possibly modified.
<i>foo</i> ^{P_k}	▷ <i>foo</i> * is evaluated as in sprogn ; see page 20.
$\underline{\textit{foo}}_2$; $\underline{\textit{bar}}_2$; $\underline{\textit{baz}}_n$	▷ Primary, secondary, and <i>n</i> th return value.
T ; NIL	▷ t , or truth in general; and nil or () .

OPTIMIZE 46
OR 20, 26, 31, 34
OTHERWISE 19, 31
OUTPUT-STREAM-P 31

PACKAGE 30
PACKAGE-ERROR 30
PACKAGE-ERROR-PACKAGE 29
PACKAGE-NAME 42
PACKAGE-NICKNAMES 42
PACKAGE-REMOVE 16
PACKAGE-SHADOWING-SYMBOLS 42
PACKAGE-USE-LIST 42
PACKAGE-USED-BY-LIST 42
PACKAGEP 41
PAIRLIS 9
PARSE-ERROR 30
PARSE-INTEGER 8
PARSE-NAMESTRING 40
PATHNAME 30, 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 32
PHASE 4
PI 3
PLUSP 3
POP 9
POSITION 13
POSITION-IF 13
POSITION-IF-NOT 13
PPRINT 34
PPRINT-DISPATCH 36
PPRINT-EXIT-IF-LIST-EXHAUSTED 35
PPRINT-FILL 35
PPRINT-INDENT 35
PPRINT-LINEAR 35
PPRINT-LOGICAL-BLOCK 35
PPRINT-NEWLINE 35
PPRINT-POP 35
PPRINT-TAB 35
PPRINT-TABULAR 35
PRESENT-SYMBOL 21
PRESENT-SYMBOLS 21
PRIN 34
PRIN1-TO-STRING 34
PRINC 34
PRINC-TO-STRING 34
PRINT 34
PRINT-NOT-READABLE 30
PRINT-NOT-READABLE-OBJECT 29
PRINT-OBJECT 34
PRINT-UNREADABLE-OBJECT 34
PROBE-FILE 41
PROCLAIM 46
PROG 20
PROG1 20
PROG2 20
PROG* 20
PROGN 20, 26
PROGRAM-ERROR 30
PROGV 16
PROVIDE 43
PSETF 16
PSETQ 16
PUSH 9
PUSHNEW 9

QUOTE 33, 45

RANDOM 4
RANDOM-STATE 30
RANDOM-STATE-P 3
RASSOC 9
RASSOC-IF 9
RASSOC-IF-NOT 9
RATIO 30, 33
RATIONAL 4, 30
RATIONALIZE 4
RATIONALP 3
READ 32
READ-BYTE 32
READ-CHAR 32
READ-CHAR-NO-HANG 32
READ-DELIMITED-LIST 32
READ-FROM-STRING 32
READ-LINE 32
READ-PRESERVING-WHITESPACE 32

READ-SEQUENCE 32
READER-ERROR 30
READTABLE 30
READTABLE-CASE 32
READTABLEP 31
REAL 30
REALP 3
REALPART 4
REDUCE 14
REINITIALIZE-INSTANCE 24
REM 4
REMF 16
REMHASH 14
REMOVE 13
REMOVE-DUPLICATES 13
REMOVE-IF 13
REMOVE-IF-NOT 13
REMOVE-METHOD 26
REMPROP 16
RENAME-FILE 41
RENAME-PACKAGE 42
REPEAT 23
REPLACE 13
REQUIRE 43
REST 8
RESTART 30
RESTART-BIND 28
RESTART-CASE 28
RESTART-NAME 29
RETURN 20, 23
RETURN-FROM 20
REVAPPEND 9
REVERSE 12
ROOM 46
ROTATEF 16
ROUND 4
ROW-MAJOR-AREF 10
RPLACA 9
RPLACD 9

SAFETY 46
SATISFIES 31
SBIT 11
SCALE-FLOAT 6
SCHAR 8
SEARCH 13
SECOND 8
SEQUENCE 30
SERIOUS-CONDITION 30
SET 16
SET-DIFFERENCE 10
SET-DISPATCH-MACRO-CHARACTER 33
SET-EXCLUSIVE-OR 10
SET-MACRO-CHARACTER 33
SET-PPRINT-DISPATCH 36
SET-SYNTAX-FROM-CHAR 32
SETF 16, 43
SETQ 16
SEVENTH 8
SHADOW 42
SHADOWING-IMPORT 42
SHARED-INITIALIZE 25
SHIFTF 16
SHORT-FLOAT 30, 33
SHORT-FLOAT-EPSILON 6
SHORT-FLOAT-NEGATIVE-EPSILON 6
SHORT-SITE-NAME 47
SIGNAL 27
SIGNED-BYTE 30
SIGNUM 4
SIMPLE-ARRAY 30
SIMPLE-BASE-STRING 30
SIMPLE-BIT-VECTOR 30
SIMPLE-BIT-VECTOR-P 10
SIMPLE-CONDITION 30
SIMPLE-CONDITION-FORMAT-ARGUMENTS 29
SIMPLE-CONDITION-FORMAT-CONTROL 29
SIMPLE-ERROR 30
SIMPLE-STRING 30
SIMPLE-STRING-P 7
SIMPLE-TYPE-ERROR 30
SIMPLE-VECTOR 30
SIMPLE-VECTOR-P 10
SIMPLE-WARNING 30
SIN 3
SINGLE-FLOAT 30, 33
SINGLE-FLOAT-EPSILON 6
SINGLE-FLOAT-NEGATIVE-EPSILON 6
SINH 3
SIXTH 8

SLEEP 20
SLOT-BOUND 23
SLOT-EXISTS-P 23
SLOT-MAKUNBOUND 24
SLOT-MISSING 25
SLOT-UNBOUND 25
SLOT-VALUE 24
SOFTWARE-TYPE 47
SOFTWARE-VERSION 47
SOME 12
SORT 12
SPACE 6, 46
SPECIAL 46
SPECIAL-OPERATOR-P 44
SPEED 46
SQRT 3
STABLE-SORT 12
STANDARD 26
STANDARD-CHAR 6, 30
STANDARD-CHAR-P 6
STANDARD-CLASS 30
STANDARD-GENERIC-FUNCTION 30
STANDARD-METHOD 30
STANDARD-OBJECT 30
STEP 45
STORE-CONDITION 30
STORE-VALUE 29
STREAM 30
STREAM-ELEMENT-TYPE 31
STREAM-ERROR 30
STREAM-ERROR-STREAM 29
STREAM-EXTERNAL-FORMAT 40
STREAMP 31
STRING 7, 30
STRING-CAPITALIZE 7
STRING-DOWNCASE 7
STRING-EQUAL 7
STRING-GREATERP 7
STRING-LEFT-TRIM 7
STRING-LESSP 7
STRING-NOT-EQUAL 7
STRING-NOT-GREATERP 7
STRING-NOT-LESSP 7
STRING-RIGHT-TRIM 7
STRING-STREAM 30
STRING-TRIM 7
STRING-UPCASE 7
STRING/= 7
STRING< 7
STRING<= 7
STRING= 7
STRING> 7
STRING>= 7
STRINGP 7
STRUCTURE 43
STRUCTURE-CLASS 30
STRUCTURE-OBJECT 30
STYLE-WARNING 30
SUBS 10
SUBSEQ 12
SUBSETP 8
SUBST 10
SUBST-IF 10
SUBST-IF-NOT 10
SUBSTITUTE 13
SUBSTITUTE-IF 13
SUBSTITUTE-IF-NOT 13
SUBTYPEP 29
SUM 23
SUMMING 23
SVREF 11
SXHASH 14
SYMBOL 21, 30, 43
SYMBOL-FUNCTION 43
SYMBOL-MACROLET 18
SYMBOL-NAME 43
SYMBOL-PACKAGE 43
SYMBOL-PLIST 43
SYMBOL-VALUE 43
SYMBOLP 41
SYMBOLS 21
SYNONYM-STREAM 30
SYNONYM-STREAM-SYMBOL 39

T 2, 30, 43
TAGBODY 20
TAILP 8
TAN 3
TANH 3
TENTH 8
TERPRI 34
THE 21, 31
THEN 21
THEREIS 23
THIRD 8
THROW 20
TIME 46
TO 21
TRACE 45

TRANSLATE-LOGICAL-PATHNAME 41
TRANSLATE-PATHNAME 41
TREE-EQUAL 10
TRUENAME 41
TRUNCATE 4
TWO-WAY-STREAM 30
TWO-WAY-STREAM-INPUT-STREAM 39
TWO-WAY-STREAM-OUTPUT-STREAM 39
TYPE 43, 46
TYPE-ERROR 30
TYPE-ERROR-DATUM 29
TYPE-ERROR-EXPECTED-TYPE 29
TYPE-OF 31
TYPECASE 31
TYPEP 29

UNBOUND-SLOT 30
UNBOUND-SLOT-INSTANCE 29
UNBOUND-VARIABLE 30
UNDEFINED-FUNCTION 30
UNEXPORT 42
UNINTERN 42
UNION 10
UNLESS 19, 23
UNREAD-CHAR 32
UNSIGNED-BYTE 30
UNTIL 23
UNTRACE 45
UNUSE-PACKAGE 42
UNWIND-PROTECT 20
UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 24
UPDATE-INSTANCE-FOR-REDEFINED-CLASS 24
UPFROM 21
UPGRADED-ARRAY-ELEMENT-TYPE 31
UPGRADED-COMPLEX-PART-TYPE 6
UPPER-CASE-P 6
UPTO 21
USE-PACKAGE 42
USE-VALUE 29
USER-HOMEDIR-PATHNAME 40
USING 21

V 38
VALUES 17, 31
VALUES-LIST 17
VARIABLE 43
VECTOR 11, 30
VECTOR-POP 11
VECTOR-PUSH 11
VECTOR-PUSH-EXTEND 11
VECTORP 10

WARN 27
WARNING 30
WHEN 19, 23
WHILE 23
WILD-PATHNAME-P 31
WITH 21
WITH-ACCESSORS 24
WITH-COMPILE-UNIT 44
WITH-CONDITION-RESTARTS 29
WITH-HASH-TABLE-ITERATOR 14
WITH-INPUT-FROM-STRING 39
WITH-OPEN-FILE 39
WITH-OPEN-STREAM 39
WITH-OUTPUT-TO-STRING 39
WITH-PACKAGE-ITERATOR 43
WITH-SIMPLE-RESTART 28
WITH-SLOTS 24
WITH-STANDARD-IO-SYNTAX 32
WRITE 35
WRITE-BYTE 34
WRITE-CHAR 34
WRITE-LINE 34
WRITE-SEQUENCE 34
WRITE-STRING 34
WRITE-TO-STRING 35

Y-OR-N-P 32
YES-OR-NO-P 32

ZEROP 3

1 Numbers

1.1 Predicates

$(f = number^+)$
 $(f \neq number^+)$
 \triangleright T if all *numbers*, or none, respectively, are equal in value.

$(f > number^+)$
 $(f \geq number^+)$
 $(f < number^+)$
 $(f \leq number^+)$
 \triangleright Return T if *numbers* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

$(f \text{ minus } a)$
 $(f \text{ zero } a)$
 $(f \text{ plus } a)$
 \triangleright T if $a < 0$, $a = 0$, or $a > 0$, respectively.

$(f \text{ even } int)$
 $(f \text{ odd } int)$
 \triangleright T if *int* is even or odd, respectively.

$(f \text{ number } foo)$
 $(f \text{ real } foo)$
 $(f \text{ rational } foo)$
 $(f \text{ float } foo)$
 $(f \text{ integer } foo)$
 $(f \text{ complex } foo)$
 $(f \text{ random-state-p } foo)$
 \triangleright T if *foo* is of indicated type.

1.2 Numeric Functions

$(f + a_{\square}^*)$
 $(f * a_{\square}^*)$
 \triangleright Return $\sum a$ or $\prod a$, respectively.

$(f - a b^*)$
 $(f / a b^*)$
 \triangleright Return $a - \sum b$ or $a / \prod b$, respectively. Without any *bs*, return $-a$ or $1/a$, respectively.

$(f 1+ a)$
 $(f 1- a)$
 \triangleright Return $a + 1$ or $a - 1$, respectively.

$(f \text{ mincf } \{ \text{mdecf} \} place [delta_{\square}])$
 \triangleright Increment or decrement the value of *place* by *delta*. Return new value.

$(f \text{ exp } p)$
 $(f \text{ exp } b p)$
 \triangleright Return e^p or b^p , respectively.

$(f \text{ log } a [b_{\square}])$
 \triangleright Return $\log_b a$ or, without *b*, $\ln a$.

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

$(f \text{ lcm } integer^*_{\square})$
 $(f \text{ gcd } integer^*_{\square})$
 \triangleright Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns 0.

cpi \triangleright long-float approximation of π , Ludolph's number.

$(f \text{ sin } a)$
 $(f \text{ cos } a)$
 $(f \text{ tan } a)$
 \triangleright $\sin a$, $\cos a$, or $\tan a$, respectively. (*a* in radians.)

$(f \text{ asin } a)$
 $(f \text{ acos } a)$
 \triangleright $\arcsin a$ or $\arccos a$, respectively, in radians.

$(f \text{ atan } a [b_{\square}])$
 \triangleright $\arctan \frac{a}{b}$ in radians.

$(f \text{ sinh } a)$
 $(f \text{ cosh } a)$
 $(f \text{ tanh } a)$
 \triangleright $\sinh a$, $\cosh a$, or $\tanh a$, respectively.

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

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

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

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

($\left. \begin{array}{l} \{f\text{round} \mid f\text{round}\} \\ \{f\text{floor} \mid f\text{ffloor}\} \\ \{f\text{ceiling} \mid f\text{fceiling}\} \\ \{f\text{truncate} \mid f\text{ftruncate}\} \end{array} \right\} n [d_{\square}])$)
▷ Return as integer or float, respectively, n/d rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.

($\left. \begin{array}{l} \{f\text{mod}\} \\ \{f\text{rem}\} \end{array} \right\} n d$)
▷ Same as *f*floor or *f*truncate, respectively, but return remainder only.

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

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

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

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

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

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

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

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

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

(*f*abs *n*) ▷ Return $|n|$.

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

(*f*float *real* [*prototype*_{□_OFF}])
▷ Convert *real* into float with type of *prototype*.

DRIBBLE 45
DYNAMIC-EXTENT 46

EACH 21
ECASE 20
ECHO-STREAM 30
ECHO-STREAM-INPUT-STREAM 39
ECHO-STREAM-OUTPUT-STREAM 39
ED 45
EIGHTH 8
ELSE 23
ELT 12
ENCODE-UNIVERSAL-TIME 47
END 23
END-OF-FILE 30
ENDP 8
ENOUGH-NAMESTRING 40
ENSURE-DIRECTORIES-EXIST 41
ENSURE-GENERIC-FUNCTION 25
EQ 15
EQL 15, 31
EQUAL 15
EQUALP 15
ERROR 27, 30
ETYPESCASE 31
EVAL 45
EVAL-WHEN 44
EVENP 3
EVERY 12
EXP 3
EXPORT 42
EXPT 3
EXTENDED-CHAR 30
EXTERNAL-SYMBOL 21
EXTERNAL-SYMBOLS 21

FBOUND 16
FCEILING 4
FDEFINITION 17
FLOOR 4
FIFTH 8
FILE-AUTHOR 41
FILE-ERROR 30
FILE-PATHNAME 29
FILE-LENGTH 41
FILE-NAMESTRING 40
FILE-POSITION 39
FILE-STREAM 30
FILE-STREAM-LENGTH 30
FILE-WRITE-DATE 41
FILL 12
FILL-POINTER 11
FINALLY 23
FIND 13
FIND-ALL-SYMBOLS 42
FIND-CLASS 24
FIND-IF 13
FIND-IF-NOT 13
FIND-METHOD 26
FIND-PACKAGE 42
FIND-RESTART 29
FIND-SYMBOL 42
FINISH-OUTPUT 39
FIRST 8
FIXNUM 30
FLET 17
FLOAT 4, 30
FLOAT-DIGITS 6
FLOAT-PRECISION 6
FLOAT-RADIX 6
FLOAT-SIGN 4
FLOATING-POINT-INEXACT 30
FLOATING-POINT-INVALID-OPERATION 30
FLOATING-POINT-OVERFLOW 30
FLOATING-POINT-UNDERFLOW 30
FLOATP 3
FLOOR 4
FMAKUNBOUND 18
FOR 21
FORCE-OUTPUT 39
FORMAT 36
FORMATTER 36
FOURTH 8
FRESH-LINE 34
FROM 21
FROUND 4
FTRUNCATE 4
FTYPE 46
FUNCALL 17
FUNCTION-KEYWORDS 26
FUNCTION-LAMBDA-EXPRESSION 17
FUNCTIONP 15

GCD 3
GENERIC-FUNCTION 30
GENSYM 43

GENTEMP 43
GET 16
GET-DECODED-TIME 47
GET-DISPATCH-MACRO-CHARACTER 33
GET-INTERNAL-REAL-TIME 46
GET-INTERNAL-RUN-TIME 46
GET-MACRO-CHARACTER 33
GET-OUTPUT-STREAM-STRING 39
GET-PROPERTIES 16
GET-SETF-EXPANSION 19
GET-UNIVERSAL-TIME 47
GETF 16
GETHASH 14
GO 20
GRAPHIC-CHAR-P 6

HANDLER-BIND 28
HANDLER-CASE 28
HASH-KEY 21
HASH-KEYS 21
HASH-TABLE 30
HASH-TABLE-COUNT 14
HASH-TABLE-P 14
HASH-TABLE-REHASH-THRESHOLD 14
HASH-TABLE-SIZE 14
HASH-TABLE-TEST 14
HASH-VALUE 21
HASH-VALUES 21
HOST-NAMESTRING 40
IDENTITY 17
IF 19, 23
IGNORABLE 46
IGNORE 46
IGNORE-ERRORS 28
IMAGPART 4
IMPORT 42
IN 21
IN-PACKAGE 42
INCF 3
INITIALIZE-INSTANCE 24
INITIALLY 23
INLINE 46
INPUT-STREAM-P 31
INSPECT 46
INTEGER 30
INTEGER-DECODE-FLOAT 6
INTEGER-LENGTH 5
INTEGERP 3
INTERACTIVE-STREAM-P 31
INTERN 42
INTERNAL-TIME-UNITS-PER-SECOND 46
INTERSECTION 10
INTO 23
INVALID-METHOD-ERROR 26
INVOKE-DEBUGGER 28
INVOKE-RESTART 29
INVOKE-RESTART-INTERACTIVELY 29
ISQRT 3
IT 23

KEYWORD 30, 41, 44
KEYWORDP 41

LABELS 17
LAMBDA-LAMBDA-LIST-KEYWORDS 19
LAMBDA-PARAMETERS-LIMIT 18
LAST 8
LCM 3
LDB 5
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 12
LET 16
LET* 16
LISP-IMPLEMENTATION-TYPE 47
LISP-IMPLEMENTATION-VERSION 47
LIST 8, 26, 30
LIST-ALL-PACKAGES 42
LIST-LENGTH 8
LIST* 8
LISTEN 39
LISTP 6
LOAD 44
LOAD-LOGICAL-PATHNAME-TRANSLATIONS 41
LOAD-TIME-VALUE 45
LOCALLY 44
LOG 3
LOGAND 5
LOGANDC1 5
LOGANDC2 5
LOGBITP 5
LOGCOUNT 5
LOGEQV 5
LOGICAL-PATHNAME-TRANSLATIONS 41
LOGIOR 5
LOGNAND 5
LOGNOR 5
LOGNOT 5
LOGORC1 5
LOGORC2 5
LOGTEST 5
LOGXOR 5
LONG-FLOAT 30, 33
LONG-FLOAT-EPSILON 6
LONG-FLOAT-NEGATIVE-EPSILON 6
LONG-SITE-NAME 47
LOOP 21
LOOP-FINISH 23
LOWER-CASE-P 6

MACHINE-INSTANCE 47
MACHINE-TYPE 47
MACHINE-VERSION 47
MACRO-FUNCTION 45
MACROEXPAND 45
MACROEXPAND-1 45
MACROLET 18
MAKE-ARRAY 10
MAKE-BROADCAST-STREAM 38
MAKE-CONCATENATED-STREAM 38
MAKE-CONDITION 27
MAKE-DISPATCH-MACRO-CHARACTER 33
MAKE-ECHO-STREAM 38
MAKE-HASH-TABLE 14
MAKE-INSTANCE 24
MAKE-INSTANCES-OBSOLETE 24
MAKE-LIST 8
MAKE-LOAD-FORM 45
MAKE-LOAD-FORM-SAVING-SLOTS 45
MAKE-METHOD 27
MAKE-PACKAGE 42
MAKE-PATHNAME 40
MAKE-RANDOM-STATE 4
MAKE-SEQUENCE 12
MAKE-STRING 7
MAKE-STRING-INPUT-STREAM 39
MAKE-STRING-OUTPUT-STREAM 39
MAKE-SYMBOL 43
MAKE-SYNONYM-STREAM 38
MAKE-TWO-WAY-STREAM 38
MAKUNBOUND 16
MAP 14

MAP-INTO 14
MAPC 9
MAPCAN 9
MAPCAR 9
MAPCON 9
MAPHASH 14
MAPL 9
MAPLIST 9
MASK-FIELD 5
MAX 4, 26
MAXIMIZE 23
MAXIMIZING 23
MEMBER 8, 31
MEMBER-IF 8
MEMBER-IF-NOT 8
MERGE 12
MERGE-PATHNAMES 40
METHOD-COMBINATION 30, 43
METHOD-COMBINATION-ERROR 26
METHOD-QUALIFIERS 26
MIN 4, 26
MINIMIZE 23
MINUSP 3
MISMATCH 12
MOD 4, 31
MOST-NEGATIVE-DOUBLE-FLOAT 6
MOST-NEGATIVE-SINGLE-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 16
MULTIPLE-VALUE-CALL 17
MULTIPLE-VALUE-LIST 17
MULTIPLE-VALUE-PROG1 20
MULTIPLE-VALUE-SETQ 16
MULTIPLE-VALUES-LIMIT 18

NAME-CHAR 7
NAMED 21
NAMESTRING 40
NBUTLAST 9
NCONC 9, 23, 26
NCONCING 23
NEVER 23
NEWLINE 6
NEXT-METHOD-P 25
NIL 2, 43
NINTERSECTION 10
NINTH 8
NO-APPLICABLE-METHOD 26
NO-NEXT-METHOD 26
NOT 15, 31, 34
NOTANY 12
NOTEVERY 12
NOTINLIN 46
NRECONC 9
NREVERSE 12
NSET-DIFFERENCE 10
NSET-EXCLUSIVE-OR 10
NSTRING-CAPITALIZE 7
NSTRING-DOWNCASE 7
NSTRING-UPCASE 7
NSUBLIS 10
NSUBST 10
NSUBST-IF 10
NSUBST-IF-NOT 10
NSUBSTITUTE 13
NSUBSTITUTE-IF 13
NSUBSTITUTE-IF-NOT 13
NTH 8
NTH-VALUE 17
NTHCDR 8
NULL 8, 30
NUMBER 30
NUMBERP 3
NUMERATOR 4
NUNION 10

ODDP 3
OF 21
OF-TYPE 21
ON 21
OPEN 38
OPEN-STREAM-P 31

Index

" 33
 ' 33
 (33
 () 43
) 33
 * 3, 30, 31, 41, 45
 ** 41, 45
 *** 45
 *BREAK-
 ON-SIGNALS* 29
 *COMPILE-FILE-
 PATHNAME* 44
 *COMPILE-FILE-
 TRUENAME* 44
 COMPILE-PRINT 44
 COMPILE-VERBOSE 44
 DEBUG-IO 40
 DEBUGGER-HOOK 29
 *DEFAULT-
 PATHNAME-
 DEFAULTS* 40
 ERROR-OUTPUT 40
 FEATURES 34
 GENSYM-COUNTER 43
 LOAD-PATHNAME 44
 LOAD-PRINT 44
 LOAD-TRUENAME 44
 LOAD-VERBOSE 44
 *MACROEXPAND-
 HOOK* 45
 MODULES 43
 PACKAGE 42
 PRINT-ARRAY 35
 PRINT-BASE 35
 PRINT-CASE 35
 PRINT-CIRCLE 36
 PRINT-ESCAPE 36
 PRINT-GENSYM 36
 PRINT-LENGTH 36
 PRINT-LEVEL 36
 PRINT-LINES 36
 *PRINT-
 MISER-WIDTH* 36
 *PRINT-PPRINT-
 DISPATCH* 36
 PRINT-PRETTY 36
 PRINT-RADIX 36
 PRINT-READABLY 36
 *PRINT-
 RIGHT-MARGIN* 36
 QUERY-IO 40
 RANDOM-STATE 4
 READ-BASE 32
 *READ-DEFAULT-
 FLOAT-FORMAT* 32
 READ-EVAL 34
 READ-SUPPRESS 33
 READTABLE 32
 STANDARD-INPUT 40
 *STANDARD-
 OUTPUT* 40
 TERMINAL-IO 40
 TRACE-OUTPUT 45
 + 3, 26, 45
 ++ 45
 +++ 45
 . 33
 @ 33
 - 3, 45
 : 33
 / 3, 33, 45
 // 45
 /// 45
 /= 3
 : 41
 :: 41
 :ALLOW-OTHER-KEYS 19
 : 33
 < 3
 <= 3
 = 3, 21
 > 3
 >= 3
 \ 34
 # 38
 #\ 33
 #| 33
 #(33
 ## 33
 #+ 34
 #- 34
 #. 34
 #: 33
 #< 34
 #= 34
 #A 33
 #B 33
 #C(33
 #O 33
 #P 33
 #R 33
 #S(33
 #X 33
 ## 34
 #| 33
 &ALLOW-
 OTHER-KEYS 19

&AUX 19
 &BODY 19
 &ENVIRONMENT 19
 &KEY 19
 &OPTIONAL 19
 &REST 19
 &WHOLE 19
 ~(~) 37
 ~* 38
 ~// 38
 ~< ~> 37
 ~< ~> 37
 ~? 38
 ~A 36
 ~B 37
 ~C 37
 ~D 37
 ~E 37
 ~F 37
 ~G 37
 ~I 37
 ~O 37
 ~P 37
 ~R 36
 ~S 36
 ~T 38
 ~W 38
 ~X 37
 ~[~] 38
 ~\$ 37
 ~% 37
 ~& 37
 ~^ 38
 ~_ 37
 ~| 37
 ~{ ~} 38
 ~~ 37
 ~↔ 37
 | 33
 | | 34
 1+ 3
 1- 3

ABORT 29
 ABOVE 21
 ABS 4
 ACONS 9
 ACOS 3
 ACOSH 4
 ACROSS 21
 ADD-METHOD 26
 ADJOIN 9
 ADJUST-ARRAY 10
 ADJUSTABLE-
 ARRAY-P 10
 ALLOCATE-INSTANCE 24
 ALPHA-CHAR-P 6
 ALPHANUMERICP 6
 ALWAYS 23
 AND
 20, 21, 23, 26, 31, 34
 APPEND 9, 23, 26
 APPENDING 23
 APPLY 17
 APROPOS 45
 APROPOS-LIST 45
 AREF 10
 ARITHMETIC-ERROR 30
 ARITHMETIC-ERROR-
 OPERANDS 29
 ARITHMETIC-ERROR-
 OPERATION 29
 ARRAY 30
 ARRAY-DIMENSION 11
 ARRAY-DIMENSION-
 LIMIT 11
 ARRAY-DIMENSIONS 11
 ARRAY-
 DISPLACEMENT 11
 ARRAY-
 ELEMENT-TYPE 31
 ARRAY-HAS-
 FILL-POINTER-P 10
 ARRAY-IN-BOUNDS-P 10
 ARRAY-RANK 11
 ARRAY-RANK-LIMIT 11
 ARRAY-ROW-
 MAJOR-INDEX 11
 ARRAY-TOTAL-SIZE 11
 ARRAY-TOTAL-
 SIZE-LIMIT 11
 ARRAYP 10
 AS 21
 ASH 5
 ASIN 3
 ASINH 4
 ASSERT 28
 ASSOC 9
 ASSOC-IF 9
 ASSOC-IF-NOT 9
 ATAN 3
 ATANH 4
 ATOM 8, 30

BASE-CHAR 30
 BASE-STRING 30
 BEING 21
 BELOW 21
 BIGNUM 30
 BIT 11, 30
 BIT-AND 11

BIT-ANDC1 11
 BIT-ANDC2 11
 BIT-EQV 11
 BIT-IOR 11
 BIT-NAND 11
 BIT-NOR 11
 BIT-NOT 11
 BIT-ORC1 11
 BIT-ORC2 11
 BIT-VECTOR 30
 BIT-VECTOR-P 10
 BIT-XOR 11
 BLOCK 20
 BOOLE 4
 BOOLE-1 4
 BOOLE-2 4
 BOOLE-AND 5
 BOOLE-ANDC1 5
 BOOLE-ANDC2 5
 BOOLE-C1 4
 BOOLE-C2 4
 BOOLE-CLR 4
 BOOLE-EQV 5
 BOOLE-IOR 5
 BOOLE-NAND 5
 BOOLE-NOR 5
 BOOLE-ORC1 5
 BOOLE-ORC2 5
 BOOLE-SET 4
 BOOLE-XOR 5
 BOOLEAN 30
 BOTH-CASE-P 6
 BOUNDP 15
 BREAK 46
 BROADCAST-STREAM 30
 BROADCAST-
 STREAM-STREAMS 39
 BUILT-IN-CLASS 30
 BUTLAST 9
 BY 21
 BYTE 5
 BYTE-POSITION 5
 BYTE-SIZE 5

CAAR 8
 CADR 8
 CALL-ARGUMENTS-
 LIMIT 18
 CALL-METHOD 27
 CALL-NEXT-METHOD 26
 CAR 8
 CASE 19
 CATCH 20
 CCASE 20
 CDAR 8
 CDDR 8
 CDR 8
 CEILING 4
 CELL-ERROR 30
 CELL-ERROR-NAME 29
 CERROR 28
 CHANGE-CLASS 24
 CHAR 8
 CHAR-CODE 7
 CHAR-CODE-LIMIT 7
 CHAR-DOWNCASE 7
 CHAR-EQUAL 6
 CHAR-GREATERP 7
 CHAR-INT 7
 CHAR-LESSP 7
 CHAR-NAME 7
 CHAR-NOT-EQUAL 6
 CHAR-NOT-GREATERP 7
 CHAR-NOT-LESSP 7
 CHAR-UPCASE 7
 CHAR/= 6
 CHAR< 6
 CHAR<= 6
 CHAR= 6
 CHAR> 6
 CHAR>= 6
 CHARACTER 7, 30, 33
 CHARACTERP 6
 CHECK-TYPE 31
 CIS 4
 CL 44
 CL-USER 44
 CLASS 30
 CLASS-NAME 24
 CLASS-OF 24
 CLEAR-INPUT 39
 CLEAR-OUTPUT 39
 CLOSE 39
 CLQR 1
 CLRHASH 14
 CODE-CHAR 7
 COERCE 31
 COLLECT 23
 COLLECTING 23
 COMMON-LISP 44
 COMMON-LISP-USER 44
 COMPILATION-SPEED 46
 COMPILE 44
 COMPILE-FILE 44
 COMPILE-
 FILE-PATHNAME 44
 COMPILED-FUNCTION 30

COMPILED-
 FUNCTION-P 44
 COMPILER-MACRO 43
 COMPILER-MACRO-
 FUNCTION 45
 COMPLEMENT 17
 COMPLEX 4, 30, 33
 COMPLEXP 3
 COMPUTE-
 APPLICABLE-
 METHODS 26
 COMPUTE-RESTARTS 29
 CONCATENATE 12
 CONCATENATED-
 STREAM 30
 CONCATENATED-
 STREAM-STREAMS 39
 COND 19
 CONDITION 30
 CONJUGATE 4
 CONS 8, 30
 CONSP 8
 CONSTANTLY 17
 CONSTANTP 15
 CONTINUE 29
 CONTROL-ERROR 30
 COPY-LIST 9
 COPY-SEQ 14
 COPY-PPRINT-
 DISPATCH 36
 COPY-READTABLE 32
 COPY-SEQ 14
 COPY-STRUCTURE 15
 COPY-SYMBOL 43
 COPY-TREE 10
 COS 3
 COSH 3
 COUNT 12, 23
 COUNT-IF 12
 COUNT-IF-NOT 12
 COUNTING 23
 CTYPESCASE 31

DEBUG 46
 DECF 3
 DECLAIM 46
 DECLARATION 46
 DECLARE 46
 DECODE-FLOAT 6
 DECODE-UNIVERSAL-
 TIME 47
 DEFCCLASS 23
 DEFCONSTANT 16
 DEFGeneric 25
 DEFINE-COMPILER-
 MACRO 18
 DEFINE-CONDITION 27
 DEFINE-METHOD-
 CDR 8
 COMBINATION 26, 27
 DEFINE-
 MODIFY-MACRO 19
 DEFINE-
 SETF-EXPANDER 19
 DEFINE-
 SYMBOL-MACRO 18
 DEFMACRO 18
 DEFMETHOD 25
 DEFPACKAGE 42
 DEFPARAMETER 16
 DEFSETF 18
 DEFSTRUCT 15
 DEFTYPE 31
 DEFUN 17
 DEFVAR 16
 DELETE 13
 DELETE-DUPLICATES 13
 DELETE-FILE 41
 DELETE-IF 13
 DELETE-IF-NOT 13
 DELETE-PACKAGE 42
 DENOMINATOR 4
 DEPOSIT-FIELD 5
 DESCRIBE 46
 DESCRIBE-OBJECT 46
 DESTRUCTURING-
 BIND 16
 DIGIT-CHAR 7
 DIGIT-CHAR-P 6
 DIRECTORY 41
 DIRECTORY-
 NAMESTRING 40
 DISASSEMBLE 46
 DIVISION-BY-ZERO 30
 DO 20, 23
 DO-ALL-SYMBOLS 43
 DO-EXTERNAL-
 SYMBOLS 43
 DO-SYMBOLS 43
 DO* 20
 DOCUMENTATION 43
 DOING 23
 DOLIST 21
 DOTIMES 21
 DOUBLE-FLOAT 30, 33
 DOUBLE-
 FLOAT-EPSILON 6
 DOUBLE-FLOAT-
 NEGATIVE-EPSILON 6
 DOWNFROM 21
 DOWNTO 21
 DPB 5

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

(flognot integer) \triangleright $\neg \text{integer}$.

(flogeqv integer*)

(flogand integer*)

\triangleright Return value of exclusive-nored or anded *integers*, respectively. Without any *integer*, return $\underline{-1}$.

(flogandc1 int-a int-b) \triangleright $\neg \text{int-a} \wedge \text{int-b}$.

(flogandc2 int-a int-b) \triangleright $\text{int-a} \wedge \neg \text{int-b}$.

(flognand int-a int-b) \triangleright $\neg(\text{int-a} \wedge \text{int-b})$.

(flogxor integer*)

(flogior integer*)

\triangleright Return value of exclusive-ored or ored *integers*, respectively. Without any *integer*, return $\underline{0}$.

(flogorc1 int-a int-b) \triangleright $\neg \text{int-a} \vee \text{int-b}$.

(flogorc2 int-a int-b) \triangleright $\text{int-a} \vee \neg \text{int-b}$.

(flognor int-a int-b) \triangleright $\neg(\text{int-a} \vee \text{int-b})$.

(flogbitp i int) \triangleright \underline{T} if zero-indexed *i*th bit of *int* is set.

(flogtest int-a int-b)

\triangleright Return \underline{T} if there is any bit set in *int-a* which is set in *int-b* as well.

(flogcount int)

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

1.4 Integer Functions

(finteger-length integer)

\triangleright Number of bits necessary to represent *integer*.

(fldb-test byte-spec integer)

\triangleright Return \underline{T} if any bit specified by *byte-spec* in *integer* is set.

(fash integer count)

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

(fldb byte-spec integer)

\triangleright Extract *byte* denoted by *byte-spec* from *integer*. **setfable**.

$\left\{ \begin{array}{l} \text{fdeposit-field} \\ \text{fdpb} \end{array} \right\} \text{int-a byte-spec int-b}$

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

(fmask-field byte-spec integer)

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

(fbyte size position)

\triangleright **Byte specifier** for a byte of *size* bits starting at a weight of *position*.

(fbyte-size byte-spec)

(fbyte-position byte-spec)

\triangleright **Size** or **position**, respectively, of *byte-spec*.

1.5 Implementation-Dependent

$\left. \begin{array}{l} \text{cshort-float} \\ \text{csingle-float} \\ \text{cdouble-float} \\ \text{clong-float} \end{array} \right\} \begin{array}{l} \text{epsilon} \\ \text{negative-epsilon} \end{array}$
 ▷ Smallest possible number making a difference when added or subtracted, respectively.

$\left. \begin{array}{l} \text{cleast-negative} \\ \text{cleast-negative-normalized} \\ \text{cleast-positive} \\ \text{cleast-positive-normalized} \end{array} \right\} \begin{array}{l} \text{short-float} \\ \text{single-float} \\ \text{double-float} \\ \text{long-float} \end{array}$
 ▷ Available numbers closest to -0 or $+0$, respectively.

$\left. \begin{array}{l} \text{cmost-negative} \\ \text{cmost-positive} \end{array} \right\} \begin{array}{l} \text{short-float} \\ \text{single-float} \\ \text{double-float} \\ \text{long-float} \\ \text{fixnum} \end{array}$
 ▷ Available numbers closest to $-\infty$ or $+\infty$, respectively.

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

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

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

(*f*upgraded-complex-part-type *foo* [*environment*_{env}])
 ▷ Type of most specialized **complex** number able to hold parts of type *foo*.

2 Characters

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

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

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

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

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

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

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

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

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

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

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

$\left. \begin{array}{l} \text{f lisp-implementation} \\ \text{f software} \\ \text{f machine} \end{array} \right\} \begin{array}{l} \text{type} \\ \text{version} \end{array}$
 ▷ Name or version of implementation, operating system, or hardware, respectively.

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

- (*f*break [*control arg**)
 ▷ Jump directly into debugger; return NIL. See page 36, *f*format, for *control* and *args*.
- (*m*time *form*)
 ▷ Evaluate *forms* and print timing information to *v**trace-output*. Return values of form.
- (*f*inspect *foo*) ▷ Interactively give information about *foo*.
- (*f*describe *foo* [*stream* [**standard-output**]])
 ▷ Send information about *foo* to *stream*.
- (*g*describe-object *foo* [*stream*])
 ▷ Send information about *foo* to *stream*. Called by *f*describe.
- (*f*disassemble *function*)
 ▷ Send disassembled representation of *function* to *v**standard-output*. Return NIL.
- (*f*room [{NIL|:default|T} [*default*]])
 ▷ Print information about internal storage management to **standard-output**.

15.4 Declarations

- (*f*proclaim *decl*)
 (*m*declare *decl**)
 ▷ Globally make declaration(s) *decl*. *decl* can be: **declaration**, **type**, **f**type, **inline**, **notinline**, **optimize**, or **special**. See below.
- (*declare* *decl**)
 ▷ Inside certain forms, locally make declarations *decl**. *decl* can be: **dynamic-extent**, **type**, **f**type, **ignorable**, **ignore**, **inline**, **notinline**, **optimize**, or **special**. See below.
- (**declaration** *foo**)
 ▷ Make *foos* names of declarations.
- (**dynamic-extent** *variable** (**function** *function**)*)
 ▷ Declare lifetime of *variables* and/or *functions* to end when control leaves enclosing block.
- (**[type]** *type variable**)
 (**f**type *type function**)
 ▷ Declare *variables* or *functions* to be of *type*.
- (**{ignorable}** {*var* (**function** *function*)}*)
 (**ignore**)
 ▷ Suppress warnings about used/unused bindings.
- (**inline** *function**)
 (**notinline** *function**)
 ▷ Tell compiler to integrate/not to integrate, respectively, called *functions* into the calling routine.
- (**optimize** {**compilation-speed** (**compilation-speed** *n*_Ⓜ)
debug (**debug** *n*_Ⓜ)
safety (**safety** *n*_Ⓜ)
space (**space** *n*_Ⓜ)
speed (**speed** *n*_Ⓜ) }*)
 ▷ Tell compiler how to optimize. *n* = 0 means unimportant, *n* = 1 is neutral, *n* = 3 means important.
- (**special** *var**) ▷ Declare *vars* to be dynamic.

16 External Environment

- (*f*get-internal-real-time)
 (*f*get-internal-run-time)
 ▷ Current time, or computing time, respectively, in clock ticks.
- c*internal-time-units-per-second
 ▷ Number of clock ticks per second.

- (*f*char-greaterp *character*+)
 (*f*char-not-lessp *character*+)
 (*f*char-lessp *character*+)
 (*f*char-not-greaterp *character*+)
 ▷ Return T if *characters* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
- (*f*char-upcase *character*)
 (*f*char-downcase *character*)
 ▷ Return corresponding uppercase/lowercase character, respectively.
- (*f*digit-char *i* [*radix*_Ⓜ]) ▷ Character representing digit *i*.
- (*f*char-name *char*) ▷ *char*'s name if any, or NIL.
- (*f*name-char *foo*) ▷ Character named *foo* if any, or NIL.
- (*f*char-int *character*)
 (*f*char-code *character*) ▷ Code of *character*.
- (*f*code-char *code*) ▷ Character with *code*.
- c*char-code-limit ▷ Upper bound of (*f*char-code *char*); ≥ 96.
- (*f*character *c*) ▷ Return #\c.

3 Strings

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

- (*f*stringp *foo*)
 (*f*simple-string-p *foo*) ▷ T if *foo* is of indicated type.
- (*f*string= {*foo bar* {**:start1** *start-foo*_Ⓜ
:start2 *start-bar*_Ⓜ
:end1 *end-foo*_Ⓜ
:end2 *end-bar*_Ⓜ }*)
 ▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.
- (*f*string{/= |**-not-equal**}
*f*string{> |**-greaterp**}
*f*string{>= |**-not-lessp**}
*f*string{< |**-lessp**}
*f*string{<= |**-not-greaterp**} } {*foo bar* {**:start1** *start-foo*_Ⓜ
:start2 *start-bar*_Ⓜ
:end1 *end-foo*_Ⓜ
:end2 *end-bar*_Ⓜ }*)
 ▷ If *foo* is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in *foo*. Otherwise return NIL. Obey/ignore, respectively, case.
- (*f*make-string *size* {**:initial-element** *char*
:element-type *type*_Ⓜ }*)
 ▷ Return string of length *size*.
- (*f*string *x*)
 (*f*string-capitalize
*f*string-upcase
*f*string-downcase } *x* {**:start** *start*_Ⓜ
:end *end*_Ⓜ }*)
 ▷ Convert *x* (**symbol**, **string**, or **character**) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.
- (*f*nstring-capitalize
*f*nstring-upcase
*f*nstring-downcase } *string* {**:start** *start*_Ⓜ
:end *end*_Ⓜ }*)
 ▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

- (*f*string-trim
*f*string-left-trim
*f*string-right-trim } *char-bag string*)
 ▷ Return string with all characters in sequence *char-bag* removed from both ends, from the beginning, or from the end, respectively.

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

(*f*parse-integer *string* $\left\{ \begin{array}{l} \text{:start } \textit{start}_{\underline{\text{NIL}}} \\ \text{:end } \textit{end}_{\underline{\text{NIL}}} \\ \text{:radix } \textit{int}_{\underline{\text{NIL}}} \\ \text{:junk-allowed } \textit{bool}_{\underline{\text{NIL}}} \end{array} \right\}$)
 ▷ Return integer parsed from *string* and index of parse end.

4 Conses

4.1 Predicates

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

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

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

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

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

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

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

4.2 Lists

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

(*f*list *foo**)
 ▷ Return list of *foos*.

(*f*list* *foo**)
 ▷ Return list of *foos* with last *foo* becoming cdr of last cons. Return *foo* if only one *foo* given.

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

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

(*f*car *list*)
 ▷ Car of *list* or NIL if *list* is NIL. **setfable**.

(*f*cdr *list*)
 (*f*rest *list*)
 ▷ Cdr of *list* or NIL if *list* is NIL. **setfable**.

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

$\left\{ \textit{ffirst} \mid \textit{fsecond} \mid \textit{fthird} \mid \textit{ffourth} \mid \textit{ffifth} \mid \textit{fsixth} \mid \dots \mid \textit{fninth} \mid \textit{ftenth} \right\}$ *list*)
 ▷ Return nth element of *list* if any, or NIL otherwise. **setfable**.

(*f*nth *n* *list*)
 ▷ Zero-indexed nth element of *list*. **setfable**.

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

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

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

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

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

(*f*make-load-form-saving-slots *foo* $\left\{ \begin{array}{l} \text{:slot-names } \textit{slots}_{\underline{\text{all local slots}}} \\ \text{:environment } \textit{environment} \end{array} \right\}$)
 ▷ Return a creation form and an initialization form which on evaluation construct an object equivalent to *foo* with *slots* initialized with the corresponding values from *foo*.

(*f*macro-function *symbol* [*environment*])
 (*f*compiler-macro-function $\left\{ \begin{array}{l} \textit{name} \\ \textit{setf } \textit{name} \end{array} \right\}$ [*environment*])

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

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

15.3 REPL and Debugging

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

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

v- ▷ Form currently being evaluated by the REPL.

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

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

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

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

$\left\{ \begin{array}{l} \textit{fmacroexpand-1} \\ \textit{fmacroexpand} \end{array} \right\}$ *form* [*environment*])
 ▷ Return macro expansion, once or entirely, respectively, of *form* and T if *form* was a macro form. Return form and NIL otherwise.

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

(*m*trace $\left\{ \begin{array}{l} \textit{function} \\ \textit{setf } \textit{function} \end{array} \right\}^*$)
 ▷ Cause *functions* to be traced. With no arguments, return list of traced functions.

(*m*untrace $\left\{ \begin{array}{l} \textit{function} \\ \textit{setf } \textit{function} \end{array} \right\}^*$)
 ▷ Stop *functions*, or each currently traced function, from being traced.

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

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

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

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

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

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

15.2 Compilation

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

▷ Return **compiled function** or replace *name*'s function definition with the compiled function. Return T in case of **warnings** or **errors**, and T in case of **warnings** or **errors** excluding **style-warnings**.

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

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

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

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

(*f*load *path* {**:verbose** *bool* [*v*load-verbose**] } {**:print** *bool* [*v*load-print**] } {**:if-does-not-exist** *bool* [T] } {**:external-format** *file-format* [*default*] }

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

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

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

*v*compile* {*print** } {*verbose** }

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

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

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

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

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

(*m*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*.

{*f*butlast *list* } [*num* [T]] ▷ *list* excluding last *num* conses.

{*f*rplaca } {*f*rplacd } *cons object*)

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

(*f*ldiff *list* *foo*)

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

(*f*adjoin *foo* *list* { {**:test** *function* [*#'eq*] } } {**:test-not** *function* } {**:key** *function* }

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

(*m*pop *place*)

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

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

(*m*pushnew *foo* *place* { {**:test** *function* [*#'eq*] } } {**:test-not** *function* } {**:key** *function* }

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

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

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

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

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

(*f*nreconc *list* *foo*)

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

{*f*mapcar } {*f*maplist } *function list*⁺)

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

{*f*mapcan } {*f*mapcon } *function list*⁺)

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

{*f*mapc } {*f*mapl } *function list*⁺)

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

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

4.3 Association Lists

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

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

(*f*acons *key value alist*)

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

{*f*assoc } {*f*rassoc } *foo alist* { {**:test** *test* [*#'eq*] } } {**:test-not** *test* } {**:key** *function* }

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

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

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

4.4 Trees

`(f tree-equal foo bar { :test test#'eq :test-not test })`

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

`({ fsubst new old tree } { :test function#'eq :test-not function :key function })`

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

`({ fsubst-if[-not] new test tree } [:key function])`

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

`({ fsublis association-list tree } { :test function#'eq :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.

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

4.5 Sets

`({ fintersection fset-difference funion fset-exclusive-or fnintersection fnset-difference fnunion fnset-exclusive-or } { :test function#'eq :test-not function :key function })`

▷ Return $a \cap b$, $a \setminus b$, $a \cup b$, or $a \Delta b$, respectively, of lists *a* and *b*.

5 Arrays

5.1 Predicates

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

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

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

5.2 Array Functions

`({ fmake-array dimension-sizes [:adjustable boolNT] } { fadjust-array array dimension-sizes { :element-type typeNT :fill-pointer {num|bool}NT :initial-element obj :initial-contents tree-or-array :displaced-to arrayNT [:displaced-index-offset iNT] } })`

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

`(f aref array [subscripts])`
 ▷ Return array element pointed to by *subscripts*. **setfable**.

`(f row-major-aref array i)`
 ▷ Return *i*th element of *array* in row-major order. **setfable**.

`({ mdo-symbols mdo-external-symbols mdo-all-symbols (var [packagev*package*] [resultNT]]) } (declare decl*)* { 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**.

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

`(f require module [pathsNT])`
 ▷ If not in **v*modules***, try *paths* to load *module* from. Signal **error** if unsuccessful. Deprecated.

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

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

14.3 Symbols

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

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

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

`(f gentemp [prefixNT] [packagev*package*]])`
 ▷ Intern fresh symbol in *package*. Deprecated.

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

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

`({ gdocumentation (setf gdocumentation) new-doc } foo { 'variable|'function 'compiler-macro 'method-combination 'structure|'type|'setf|'T })`

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

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

cnil_c()
 ▷ Falsity; the empty list; the empty type, subtype of every type; **v*standard-input***; **v*standard-output***; the global environment.

```
(mdefpackage 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.

```
(fmake-package foo {
  (:nicknames (nick*)NIL)
  (:use (used-package*)*)
})
```

▷ Create package *foo*.

```
(frename-package package new-name [new-nicknamesNIL])
```

▷ Rename *package*. Return renamed package.

```
(min-package foo) ▷ Make package foo current.
```

```
{
  (fuse-package)
  (funuse-package)
} other-packages [packagev*package*])
```

▷ Make exported symbols of *other-packages* available in *package*, or remove them from *package*, respectively. Return T.

```
(fpackage-use-list package)
```

```
(fpackage-used-by-list package)
```

▷ List of other packages used by/using *package*.

```
(fdelete-package package)
```

▷ Delete *package*. Return T if successful.

```
v*package*common-lisp-user
```

▷ The current package.

```
(flist-all-packages)
```

▷ List of registered packages.

```
(fpackage-name package)
```

▷ Name of *package*.

```
(fpackage-nicknames package)
```

▷ Nicknames of *package*.

```
(ffind-package name)
```

▷ Package with *name* (case-sensitive).

```
(ffind-all-symbols foo)
```

▷ List of symbols *foo* from all registered packages.

```
{
  (fintern)
  (ffind-symbol)
} foo [packagev*package*])
```

▷ Intern or find, respectively, symbol *foo* in *package*. Second return value is one of :internal, :external, or :inherited (or NIL if *f*intern has created a fresh symbol).

```
(funintern symbol [packagev*package*])
```

▷ Remove *symbol* from *package*, return T on success.

```
{
  (fimport)
  (fshadowing-import)
} symbols [packagev*package*])
```

▷ Make *symbols* internal to *package*. Return T. In case of a name conflict signal correctable **package-error** or shadow the old symbol, respectively.

```
(fshadow symbols [packagev*package*])
```

▷ Make *symbols* of *package* shadow any otherwise accessible, equally named symbols from other packages. Return T.

```
(fpackage-shadowing-symbols package)
```

▷ List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.

```
(fexport symbols [packagev*package*])
```

▷ Make *symbols* external to *package*. Return T.

```
(funexport symbols [packagev*package*])
```

▷ Revert *symbols* to internal status. Return T.

```
(farray-row-major-index array [subscripts])
```

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

```
(farray-dimensions array)
```

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

```
(farray-dimension array i)
```

▷ Length of *i*th dimension of *array*.

```
(farray-total-size array)
```

▷ Number of elements in *array*.

```
(farray-rank array)
```

▷ Number of dimensions of *array*.

```
(farray-displacement array)
```

▷ Target array and offset.

```
(fbit bit-array [subscripts])
```

```
(fsbit simple-bit-array [subscripts])
```

▷ Return element of *bit-array* or of *simple-bit-array*. **setf**able.

```
(fbit-not bit-array [result-bit-arrayNIL])
```

▷ 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.

```
{
  (fbit-eqv)
  (fbit-and)
  (fbit-andc1)
  (fbit-andc2)
  (fbit-nand)
  (fbit-ior)
  (fbit-orc1)
  (fbit-orc2)
  (fbit-xor)
  (fbit-nor)
} bit-array-a bit-array-b [result-bit-arrayNIL])
```

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

```
carray-rank-limit
```

▷ Upper bound of array rank; ≥ 8 .

```
carray-dimension-limit
```

▷ Upper bound of an array dimension; ≥ 1024 .

```
carray-total-size-limit
```

▷ Upper bound of array size; ≥ 1024 .

5.3 Vector Functions

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

```
(fvector foo*)
```

▷ Return fresh simple vector of *foos*.

```
(fsvref vector i)
```

▷ Element *i* of simple *vector*. **setf**able.

```
(fvector-push foo vector)
```

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

```
(fvector-push-extend foo vector [num])
```

▷ Replace element of *vector* pointed to by fill pointer with *foo*, then increment fill pointer. Extend *vector*'s size by \geq *num* if necessary.

```
(fvector-pop vector)
```

▷ Return element of *vector* its fillpointer points to after decrementation.

```
(ffill-pointer vector)
```

▷ Fill pointer of *vector*. **setf**able.

6 Sequences

6.1 Sequence Predicates

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

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

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

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

$(f\text{mismatch}\ sequence\text{-}a\ sequence\text{-}b\ \left\{ \begin{array}{l} \text{:from-end}\ bool\ \underline{\text{NIL}} \\ \text{:test}\ function\ \underline{\neq\ \text{NIL}} \\ \text{:test-not}\ function \\ \text{:start1}\ start\text{-}a\ \underline{\text{NIL}} \\ \text{:start2}\ start\text{-}b\ \underline{\text{NIL}} \\ \text{:end1}\ end\text{-}a\ \underline{\text{NIL}} \\ \text{:end2}\ end\text{-}b\ \underline{\text{NIL}} \\ \text{:key}\ function \end{array} \right\})$

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

6.2 Sequence Functions

$(f\text{make-sequence}\ sequence\text{-}type\ size\ [:\text{initial-element}\ foo])$

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

$(f\text{concatenate}\ type\ sequence^*)$

▷ Return concatenated sequence of *type*.

$(f\text{merge}\ type\ \widetilde{sequence\text{-}a}\ \widetilde{sequence\text{-}b}\ test\ [:\text{key}\ function\ \underline{\text{NIL}}])$

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

$(f\text{fill}\ \widetilde{sequence}\ foo\ \left\{ \begin{array}{l} \text{:start}\ start\ \underline{\text{NIL}} \\ \text{:end}\ end\ \underline{\text{NIL}} \end{array} \right\})$

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

$(f\text{length}\ sequence)$

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

$(f\text{count}\ foo\ sequence\ \left\{ \begin{array}{l} \text{:from-end}\ bool\ \underline{\text{NIL}} \\ \text{:test}\ function\ \underline{\neq\ \text{NIL}} \\ \text{:test-not}\ function \\ \text{:start}\ start\ \underline{\text{NIL}} \\ \text{:end}\ end\ \underline{\text{NIL}} \\ \text{:key}\ function \end{array} \right\})$

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

$\left\{ \begin{array}{l} \text{count-if} \\ \text{count-if-not} \end{array} \right\} test\ sequence\ \left\{ \begin{array}{l} \text{:from-end}\ bool\ \underline{\text{NIL}} \\ \text{:start}\ start\ \underline{\text{NIL}} \\ \text{:end}\ end\ \underline{\text{NIL}} \\ \text{:key}\ function \end{array} \right\})$

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

$(f\text{elt}\ sequence\ index)$

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

$(f\text{subseq}\ sequence\ start\ [end\ \underline{\text{NIL}}])$

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

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

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

$(f\text{reverse}\ sequence)$

$(f\text{nreverse}\ \widetilde{sequence})$

▷ Return sequence in reverse order.

$(f\text{translate-pathname}\ path\text{-or-stream}\ wildcard\text{-}path\text{-}a\ wildcard\text{-}path\text{-}b)$

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

$(f\text{pathname}\ path\text{-or-stream})$ ▷ Pathname of *path-or-stream*.

$(f\text{logical-pathname}\ logical\text{-}path\text{-or-stream})$

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

$(f\text{logical-pathname-translations}\ logical\text{-}host)$

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

$(f\text{load-logical-pathname-translations}\ logical\text{-}host)$

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

$(f\text{translate-logical-pathname}\ path\text{-or-stream})$

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

$(f\text{probe-file}\ file)$

$(f\text{truename}\ file)$

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

$(f\text{file-write-date}\ file)$

▷ Time at which *file* was last written.

$(f\text{file-author}\ file)$

▷ Return name of file owner.

$(f\text{file-length}\ stream)$

▷ Return length of stream.

$(f\text{rename-file}\ foo\ bar)$

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

$(f\text{delete-file}\ file)$

▷ Delete *file*. Return T.

$(f\text{directory}\ path)$

▷ List of pathnames matching *path*.

$(f\text{ensure-directories-exist}\ path\ [:\text{verbose}\ bool])$

▷ Create parts of *path* if necessary. Second return value is T if something has been created.

14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see **loop**, page 21.

14.1 Predicates

$(f\text{symbolp}\ foo)$

$(f\text{packagep}\ foo)$

$(f\text{keywordp}\ foo)$

▷ T if *foo* is of indicated type.

14.2 Packages

$:\text{bar}|\text{keyword}:\text{bar}$

▷ Keyword, evaluates to :bar.

$\text{package}:\text{symbol}$

▷ Exported *symbol* of *package*.

$\text{package}::\text{symbol}$

▷ Possibly unexported *symbol* of *package*.

- (*f*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.

13.7 Pathnames and Files

(*f*make-pathname

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

▷ Construct a logical pathname if there is a logical pathname translation for *host*, otherwise construct a physical pathname. For **:case** **:local**, leave case of components unchanged. For **:case** **:common**, leave mixed-case components unchanged; convert all-uppercase components into local customary case; do the opposite with all-lowercase components.

<i>f</i> pathname-host <i>f</i> pathname-device <i>f</i> pathname-directory <i>f</i> pathname-name <i>f</i> pathname-type	}	<i>path-or-stream</i> [:case {: <i>local</i> : <i>common</i> : <i>local</i> }]
---	---	--

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

(*f*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.

(*f*merge-pathnames *path-or-stream* [*default-path-or-stream* √*default-pathname-defaults*]
 [*default-version* *newest*])
 ▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

√*default-pathname-defaults*
 ▷ Pathname to use if one is needed and none supplied.

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

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

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

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

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

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

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

{ <i>f</i> find : <i>f</i> position} <i>foo sequence</i>	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>test</i> <i>function</i> _{#='eq} : <i>test-not</i> <i>test</i> : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> }
---	---	--

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

{ <i>f</i> find-if : <i>f</i> find-if-not : <i>f</i> position-if : <i>f</i> position-if-not} <i>test sequence</i>	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> }
--	---	---

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

(<i>f</i> search <i>sequence-a</i> <i>sequence-b</i>)	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>test</i> <i>function</i> _{#='eq} : <i>test-not</i> <i>function</i> : <i>start1</i> <i>start-a</i> ₀ : <i>start2</i> <i>start-b</i> ₀ : <i>end1</i> <i>end-a</i> _{NIL} : <i>end2</i> <i>end-b</i> _{NIL} : <i>key</i> <i>function</i> }
---	---	--

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

{ <i>f</i> remove <i>foo sequence</i> : <i>f</i> delete <i>foo sequence</i> }	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>test</i> <i>function</i> _{#='eq} : <i>test-not</i> <i>function</i> : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> : <i>count</i> <i>count</i> _{NIL} }
--	---	--

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

{ <i>f</i> remove-if : <i>f</i> remove-if-not : <i>f</i> delete-if : <i>f</i> delete-if-not} <i>test sequence</i>	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> : <i>count</i> <i>count</i> _{NIL} }
--	---	---

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

{ <i>f</i> remove-duplicates <i>sequence</i> : <i>f</i> delete-duplicates <i>sequence</i> }	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>test</i> <i>function</i> _{#='eq} : <i>test-not</i> <i>function</i> : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> }
--	---	--

▷ Make copy of sequence without duplicates.

{ <i>f</i> substitute <i>new old sequence</i> : <i>f</i> nsubstitute <i>new old sequence</i> }	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>test</i> <i>function</i> _{#='eq} : <i>test-not</i> <i>function</i> : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> : <i>count</i> <i>count</i> _{NIL} }
---	---	--

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

{ <i>f</i> substitute-if : <i>f</i> substitute-if-not : <i>f</i> nsubstitute-if : <i>f</i> nsubstitute-if-not} <i>new test sequence</i>	}	{: <i>from-end</i> <i>bool</i> _{NIL} : <i>start</i> <i>start</i> ₀ : <i>end</i> <i>end</i> _{NIL} : <i>key</i> <i>function</i> : <i>count</i> <i>count</i> _{NIL} }
--	---	---

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

(<i>f</i> replace <i>sequence-a</i> <i>sequence-b</i>)	}	{: <i>start1</i> <i>start-a</i> ₀ : <i>start2</i> <i>start-b</i> ₀ : <i>end1</i> <i>end-a</i> _{NIL} : <i>end2</i> <i>end-b</i> _{NIL} }
--	---	---

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

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

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

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

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

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

7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see **loop**, page 21.

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

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

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

▷ Make a hash table.

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

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

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

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

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

(*mwith-hash-table-iterator* (*foo hash-table*) (**declare** $\widehat{\text{decl}}^*$)^{*} *form*^{P_k})
 ▷ Return values of forms. In *forms*, invocations of (*foo*) return: T if an entry is returned; its key; its value.

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

(*fhash-table-size* *hash-table*)

(*fhash-table-rehash-size* *hash-table*)

(*fhash-table-rehash-threshold* *hash-table*)

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

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

▷ Return stream of indicated type.

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

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

(*fconcatenated-stream-streams* *concatenated-stream*)

(*fbroadcast-stream-streams* *broadcast-stream*)

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

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

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

(*fecho-stream-input-stream* *echo-stream*)

(*fecho-stream-output-stream* *echo-stream*)

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

(*fsynonym-stream-symbol* *synonym-stream*)

▷ Return symbol of *synonym-stream*.

(*fget-output-stream-string* *string-stream*)

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

(*ffile-position* *stream* $\left\{ \begin{array}{l} \text{:start} \\ \text{:end} \\ \text{:position} \end{array} \right\}$)

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

(*ffile-string-length* *stream* *foo*)

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

(*flisten* [*stream* v*standard-input*])

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

(*fclear-input* [*stream* v*standard-input*])

▷ Clear input from *stream*, return NIL.

$\left\{ \begin{array}{l} \text{fclear-output} \\ \text{fforce-output} \\ \text{ffinish-output} \end{array} \right\}$ [*stream* v*standard-output*])

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

(*fclose* *stream* [*:abort* bool NIL])

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

(*mwith-open-file* (*stream path open-arg*^{*}) (**declare** $\widehat{\text{decl}}^*$)^{*} *form*^{P_k})

▷ Use *fopen* with *open-args* to temporarily create *stream* to *path*; return values of forms.

(*mwith-open-stream* (*foo stream*) (**declare** $\widehat{\text{decl}}^*$)^{*} *form*^{P_k})

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

(*mwith-input-from-string* (*foo string* $\left\{ \begin{array}{l} \text{:index } \text{index} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \end{array} \right\}$) (**declare**

$\widehat{\text{decl}}^*$)^{*} *form*^{P_k})

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

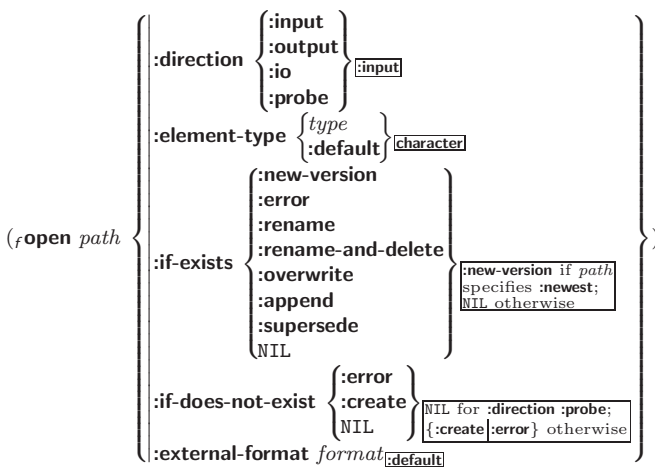
(*mwith-output-to-string* (*foo* [*string* NIL] [*:element-type* *type* character])

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

▷ 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.

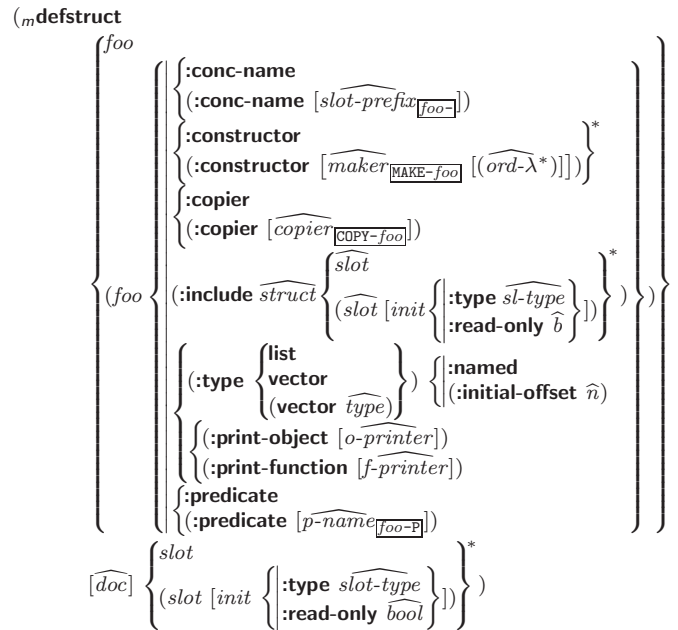
- ~ [c₀] [i₀] [:] [C] T
 ▷ **Tabulate.** Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible. With $:$, calculate column numbers relative to the immediately enclosing section. With C , move to column number $c_0 + c + ki$ where c_0 is the current position.
- {~ [m₀] *|~ [m₀] :*|~ [n₀] C*}
 ▷ **Go-To.** Jump m arguments forward, or backward, or to argument n .
- ~ [limit] [:] [C] { text ~}
 ▷ **Iteration.** Use *text* repeatedly, up to *limit*, as control string for the elements of the list argument or (with C) for the remaining arguments. With $:$ or C , 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 $\sim < \sim >$, $\sim < \sim : >$, $\sim \{ \sim \}$, $\sim ?$, or the entire f format operation. With one to three prefixes, act only if $x = 0$, $x = y$, or $x \leq y \leq z$, respectively.
- ~ [i] [:] [C] [[text ~;]* text] [~:: default] ~]
 ▷ **Conditional Expression.** Use the zero-indexed argument (or i th if given) *text* as a f format control subclause. With $:$, use the first *text* if the argument value is NIL, or the second *text* if it is T. With C , do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.
- {~?|~C?}
 ▷ **Recursive Processing.** Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.
- ~ [prefix {,prefix}*] [:] [C] / [package [:] :cl-user] function/
 ▷ **Call Function.** Call all-uppercase *package::function* with the arguments stream, format-argument, colon-p, at-sign-p and *prefixes* for printing format-argument.
- ~ [:] [C] W
 ▷ **Write.** Print argument of any type obeying every printer control variable. With $:$, pretty-print. With C , print without limits on length or depth.
- {V|#}
 ▷ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams



- (f make-concatenated-stream input-stream*)
 (f make-broadcast-stream output-stream*)
 (f make-two-way-stream input-stream-part output-stream-part)
 (f make-echo-stream from-input-stream to-output-stream)
 (f make-synonym-stream variable-bound-to-stream)

8 Structures



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

9 Control Structure

9.1 Predicates

- (f eq foo bar) ▷ T if *foo* and *bar* are identical.
- (f eql foo bar)
 ▷ T if *foo* and *bar* are identical, or the same **character**, or **numbers** of the same type and value.
- (f equal foo bar)
 ▷ T if *foo* and *bar* are f **eql**, or are equivalent **pathnames**, or are **conses** with f **equal** cars and cdrs, or are **strings** or **bit-vectors** with f **eql** elements below their fill pointers.
- (f equalp foo bar)
 ▷ T if *foo* and *bar* are identical; or are the same **character** ignoring case; or are **numbers** of the same value ignoring type; or are equivalent **pathnames**; or are **conses** or **arrays** of the same shape with f **equalp** elements; or are structures of the same type with f **equalp** elements; or are **hash-tables** of the same size with the same **:test** function, the same keys in terms of **:test** function, and f **equalp** elements.
- (f not foo) ▷ T if *foo* is NIL; **NIL** otherwise.
- (f boundp symbol) ▷ T if *symbol* is a special variable.
- (f constantp foo [environment_{env}])
 ▷ T if *foo* is a constant form.
- (f functionp foo) ▷ T if *foo* is of type **function**.

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

9.2 Variables

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

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

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

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

(*fset* \widehat{symbol} *foo*) \triangleright Set *symbol*'s value cell to *foo*. Deprecated.

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

(*mshiftf* \widehat{place}^+ *foo*)
 \triangleright Store value of *foo* in rightmost *place* shifting values of *places* left, returning first *place*.

(*mrotatef* \widehat{place}^*)
 \triangleright Rotate values of *places* left, old first becoming new last *place*'s value. Return NIL.

(*fmakunbound* \widehat{foo}) \triangleright Delete special variable *foo* if any.

(*fget* *symbol* *key* [*default* \square])
(*fgetf* *place* *key* [*default* \square])
 \triangleright First entry *key* from property list stored in *symbol*/in *place*, respectively, or *default* if there is no *key*. setfable.

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

(*fremprop* \widehat{symbol} *key*)
(*mremf* *place* *key*)
 \triangleright Remove first entry *key* from property list stored in *symbol*/in *place*, respectively. Return T if *key* was there, or NIL otherwise.

(*sprog* *symbols* *values* *form*^P)
 \triangleright Evaluate *forms* with locally established dynamic bindings of *symbols* to *values* or NIL. Return values of *forms*.

($\left\{ \begin{array}{l} \text{(slet)} \\ \text{(slet*)} \end{array} \right\}$ ($\left\{ \begin{array}{l} \text{(name } \widehat{value} \square) \\ \text{(name } \widehat{value} \square) \end{array} \right\}^*$) (*declare* \widehat{decl}^*)^P *form*^P)
 \triangleright Evaluate *forms* with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return values of *forms*.

(*mmultiple-value-bind* (\widehat{var}^*) *values-form* (*declare* \widehat{decl}^*)^P *body-form*^P)
 \triangleright Evaluate *body-forms* with *vars* lexically bound to the return values of *values-form*. Return values of *body-forms*.

(*mdestructuring-bind* *destruct- λ* *bar* (*declare* \widehat{decl}^*)^P *form*^P)
 \triangleright 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.

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

\sim [*width*] [*'dec-digits*] [*'shift* \square] [*'overflow-char*] [*'pad-char* \square] [:] [**@**] **F**
 \triangleright **Fixed-Format Floating-Point**. With **@**, always prepend a sign.

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

\sim [*dec-digits* \square] [*'int-digits* \square] [*'width* \square] [*'pad-char* \square] [:] [**@**] **S**
 \triangleright **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**}
 \triangleright **Character**. Print, spell out, print in **#** syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

{**~(text ~)|~:(text ~)|~@ (text ~)|~@: (text ~)**}
 \triangleright **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**}
 \triangleright **Plural**. If argument **eq1** print nothing, otherwise print **s**; do the same for the previous argument; if argument **eq1** print **y**, otherwise print **ies**; do the same for the previous argument, respectively.

\sim [*n* \square] **%** \triangleright **Newline**. Print *n* newlines.

\sim [*n* \square] **&**
 \triangleright **Fresh-Line**. Print *n* – 1 newlines if output stream is at the beginning of a line, or *n* newlines otherwise.

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

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

\sim [*n* \square] **|** \triangleright **Page**. Print *n* page separators.

\sim [*n* \square] **~** \triangleright **Tilde**. Print *n* tildes.

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

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

{ \sim [*n* \square] **i** | \sim [*n* \square] **:i**}
 \triangleright **Indent**. Set indentation to *n* relative to leftmost/to current position.

- v*print-circle***_[NIL]
 ▷ If T, avoid indefinite recursion while printing circular structure.
- v*print-escape***_[NIL]
 ▷ If NIL, do not print escape characters and package prefixes.
- v*print-gensym***_[NIL] ▷ If T, print #: before uninterned symbols.
- v*print-length***_[NIL]
v*print-level*_[NIL]
v*print-lines*_[NIL]
 ▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.
- v*print-miser-width***
 ▷ If integer and greater than the width available for printing a substructure, switch to the more compact miser style.
- v*print-pretty*** ▷ If T, print prettily.
- v*print-radix***_[NIL] ▷ If T, print rationals with a radix indicator.
- v*print-readably***_[NIL]
 ▷ If T, print *f*readably or signal error **print-not-readable**.
- v*print-right-margin***_[NIL]
 ▷ Right margin width in ems while pretty-printing.
- (*f*set-pprint-dispatch *type function* [*priority*]_[NIL]
 [*table*_[v*print-pprint-dispatch*]])
 ▷ Install entry comprising *function* of arguments stream and object to print; and *priority* as *type* into *table*. If *function* is NIL, remove *type* from *table*. Return NIL.
- (*f*pprint-dispatch *foo* [*table*_[v*print-pprint-dispatch*]])
 ▷ Return highest priority *function* associated with type of *foo* and T if there was a matching type specifier in *table*.
- (*f*copy-pprint-dispatch [*table*_[v*print-pprint-dispatch*]])
 ▷ Return copy of *table* or, if *table* is NIL, initial value of **v*print-pprint-dispatch***.
- v*print-pprint-dispatch*** ▷ Current pretty print dispatch table.

13.5 Format

- (*m*formatter *control*)
 ▷ Return function of *stream* and *arg** applying *f*format to *stream*, *control*, and *arg** returning NIL or any excess *args*.
- (*f*format {T|NIL|out-string|out-stream} *control arg**)
 ▷ Output string *control* which may contain ~ directives possibly taking some *args*. Alternatively, *control* can be a function returned by *m*formatter which is then applied to *out-stream* and *arg**. Output to *out-string*, *out-stream* or, if first argument is T, to **v*standard-output***. Return NIL. If first argument is NIL, return formatted output.
- ~ [*min-col*]_[0] [, [*col-inc*]_[0] [, [*min-pad*]_[0] [, [*'pad-char*]_[]]]
 [:] [**@**] {**A**|**S**}
 ▷ **Aesthetic/Standard**. Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with **@**, add *pad-chars* on the left rather than on the right.
- ~ [*radix*]_[0] [, [*width*] [, [*'pad-char*]_[] [, [*'comma-char*]_[] [, [*comma-interval*]_[]]] [:] [**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.

9.3 Functions

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

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

$$[\&key \left\{ \left\{ (var [init_{NIL}] [supplied-p]) \right\}^* \right\} [\&allow-other-keys]]$$

$$[\&aux \left\{ (var [init_{NIL}]) \right\}^*])$$

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

$$\left(\left\{ m\text{defun} \left\{ \begin{array}{l} foo (ord-\lambda^*) \\ (\text{setf } foo) (new-value ord-\lambda^*) \end{array} \right\} \right\} \left\{ \left(\widehat{\text{declare } decl^*} \right)^* \right\} \right)$$

$$\left(\left\{ m\text{lambda} \left\{ \begin{array}{l} (ord-\lambda^*) \\ form^* \end{array} \right\} \right\} \right)$$

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

$$\left(\left\{ \begin{array}{l} \text{flet} \\ \text{slabels} \end{array} \right\} \left(\left\{ \begin{array}{l} foo (ord-\lambda^*) \\ (\text{setf } foo) (new-value ord-\lambda^*) \end{array} \right\} \right) \right)$$

$$\left(\left\{ \left(\widehat{\text{declare } local-decl^*} \right)^* \right\} local-form^* \right) (\widehat{\text{declare } decl^*})^*$$

$$form^*$$

▷ 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 **sblock** around its corresponding *local-form**. Only for **slabels**, functions *foo* are visible inside *local-forms*. Return values of forms.

$$\left(\text{sfunction} \left\{ \begin{array}{l} foo \\ (m\text{lambda } form^*) \end{array} \right\} \right)$$

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

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

▷ Values of function called with *args* and the list elements of *args*. **setfable** if *function* is one of *f*aref, *f*bit, and *f*sbit.

(*f*funcall *function arg**) ▷ Values of function called with *args*.

(*s*multiple-value-call *function form**)

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

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

(*f*values *foo**)

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

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

(*m*nth-value *n form*)

▷ Zero-indexed *n*th return value of *form*.

(*f*complement *function*)

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

(*f*constantly *foo*)

▷ Function of any number of arguments returning *foo*.

(*f*identity *foo*) ▷ Return *foo*.

(*f*function-lambda-expression *function*)

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

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

▷ Definition of global function *foo*. **setfable**.

(*f*fmakunbound *foo*)

▷ Remove global function or macro definition *foo*.

ccall-arguments-limit

clambda-parameters-limit

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

cmultiple-values-limit

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

9.4 Macros

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

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

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

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

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

[&allow-other-keys] [&aux {*var* (*macro-λ**)}*] [*E*]

or

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

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

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

{*m*defmacro {*f*define-compiler-macro} {*foo* (*setf* *foo*)} (*macro-λ**)

{(declare *decl**)*} *form*^{P*}
doc

▷ Define macro *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** named *foo*.

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

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

(*s*macrolet ((*foo* (*macro-λ**) { (declare *local-decl**)* }
doc)

macro-form^{P*}*) (declare *decl**)* *form*^{P*})

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

(*s*symbol-macrolet ((*foo expansion-form*)* (declare *decl**)* *form*^{P*})

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

(*m*defsetf *function* {*updater* [*doc*]} {*setf-λ** (*s-var*)*} { (declare *decl**)* } *form*^{P*} }

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

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

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

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

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

(*f*pprint-fill *stream* *foo* [*parenthesis*] [*noop*])

(*f*pprint-tabular *stream* *foo* [*parenthesis*] [*noop*] [*n*])

(*f*pprint-linear *stream* *foo* [*parenthesis*] [*noop*])

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

(*m*pprint-logical-block (*stream* *list* { {*prefix* *string*}
{*per-line-prefix* *string*}
{*suffix* *string*}})

(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 *f*write. Return NIL.

(*m*pprint-pop)

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

(*f*pprint-tab {*line*
:*line-relative*
:*section*
:*section-relative*} *c* *i* [*stream* *v**standard-output*])

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

(*f*pprint-indent {*block*
:*current*} *n* [*stream* *v**standard-output*])

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

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

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

(*f*pprint-newline {*linear*
:*fill*
:*miser*
:*mandatory*} [*stream* *v**standard-output*])

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

*v**print-array* ▷ If T, print arrays *f*readably.

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

*v**print-case*_{uppercase}

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

- `#.form` ▷ Read-time value of *form*.
- `√*read-eval*□` ▷ If NIL, a **reader-error** is signalled at `#.`.
- `#integer= foo` ▷ Give *foo* the label *integer*.
- `#integer#` ▷ Object labelled *integer*.
- `#<` ▷ Have the reader signal **reader-error**.
- `#+feature when-feature`
`#-feature unless-feature`
 ▷ Means *when-feature* if *feature* is T; means *unless-feature* if *feature* is NIL. *feature* is a symbol from `√*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.

13.4 Printer

`{fprin1
fprint
fpprint
fprinc}` *foo* [*stream* `√*standard-output*`]

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

`(fprin1-to-string foo)`

`(fprinc-to-string foo)`

▷ Print *foo* to *string* *f*readably or human-readably, respectively.

`(gprint-object object stream)`

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

`(mprint-unreadable-object (foo stream {:type bool□□□
:identity bool□□□}) formpk)`

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

`(fterpri [stream √*standard-output*])`

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

`(ffresh-line [stream √*standard-output*])`

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

`(fwrite-char char [stream √*standard-output*])`

▷ Output *char* to *stream*.

`{fwrite-string
fwrite-line}` *string* [*stream* `√*standard-output*`] [`{:start start□
:end end□□□}`]

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

`(fwrite-byte byte stream)`

▷ Write *byte* to binary *stream*.

`(fwrite-sequence sequence stream {:start start□
:end end□□□})`

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

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

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

`(mdefine-setf-expander function (macro-λ*) {(declare decl*)
doc})`

form^{pk})

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

`(fget-setf-expansion place [environment □□□])`

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

`(mdefine-modify-macro foo ([&optional`

var `{var [init □□□ [supplied-p]]}`) [`&rest` *var*]) *function* [*doc*])

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

λlambda-list-keywords

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

&whole *var*

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

&optional *var**

▷ Bind *vars* to corresponding arguments if any.

`{&rest` **&body** *var*

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

&key *var**

▷ Bind *vars* to corresponding keyword arguments.

&allow-other-keys

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

&environment *var*

▷ Bind *var* to the lexical compilation environment.

&aux *var**

▷ Bind *vars* as in **!let***.

9.5 Control Flow

`(sif test then [else □□□])`

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

`(mcond (test thenpk [test])*)`

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

`{mwhen
munless}` *test* *foo*^{pk})

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

`(mcase test {(key*)
key} foopk)* [{T
otherwise} barpk □□□])`

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

$\left\{ \begin{array}{l} \text{m} \\ \text{ccase} \end{array} \right\} \text{test} \left(\left\{ \begin{array}{l} \widehat{\text{key}} \\ \text{key} \end{array} \right\} \text{foo}^{\text{P}} \right)^*$
 ▷ Return the values of the first *foo* one of whose *keys* is **eq** *test*. Signal non-correctable/correctable **type-error** if there is no matching *key*.

$(\text{m} \text{and} \text{form}^* \text{P})$
 ▷ Evaluate *forms* from left to right. Immediately return **NIL** if one *form*'s value is **NIL**. Return values of last *form* otherwise.

$(\text{m} \text{or} \text{form}^* \text{NIL})$
 ▷ Evaluate *forms* from left to right. Immediately return primary value of first non-**NIL**-evaluating form, or all values if last *form* is reached. Return **NIL** if no *form* returns **T**.

$(\text{s} \text{progn} \text{form}^* \text{NIL})$
 ▷ Evaluate *forms* sequentially. Return values of last *form*.

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

$\left\{ \begin{array}{l} \text{m} \\ \text{prog} \end{array} \right\} \left(\left\{ \begin{array}{l} \text{name} \\ (\text{name} \text{value}_{\text{NIL}}) \end{array} \right\}^* \right) (\text{declare} \widehat{\text{decl}}^*) \left\{ \begin{array}{l} \widehat{\text{tag}} \\ \text{form} \end{array} \right\}^*$
 ▷ Evaluate $\text{s} \text{tagbody}$ -like body with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return **NIL** or explicitly *m*returned values. Implicitly, the whole form is a $\text{s} \text{block}$ named **NIL**.

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

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

$(\text{s} \text{return-from} \text{foo} [\text{result}_{\text{NIL}}])$
 $(\text{m} \text{return} [\text{result}_{\text{NIL}}])$
 ▷ Have nearest enclosing $\text{s} \text{block}$ named *foo*/named **NIL**, respectively, return with values of *result*.

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

$(\text{s} \text{go} \widehat{\text{tag}})$
 ▷ Within the innermost possible enclosing $\text{s} \text{tagbody}$, jump to a tag $\text{r} \text{eq} \widehat{\text{tag}}$.

$(\text{s} \text{catch} \text{tag} \text{form}^{\text{P}})$
 ▷ Evaluate *forms* and return their values unless interrupted by $\text{s} \text{throw}$.

$(\text{s} \text{throw} \text{tag} \text{form})$
 ▷ Have the nearest dynamically enclosing $\text{s} \text{catch}$ with a tag $\text{r} \text{eq} \text{tag}$ return with the values of *form*.

$(\text{r} \text{sleep} n)$ ▷ Wait *n* seconds; return **NIL**.

9.6 Iteration

$\left\{ \begin{array}{l} \text{m} \\ \text{do} \end{array} \right\} \left\{ \begin{array}{l} \text{var} \\ (\text{var} [\text{start} [\text{step}]] \end{array} \right\}^* (\text{stop} \text{result}^{\text{P}}) (\text{declare} \widehat{\text{decl}}^*) \left\{ \begin{array}{l} \widehat{\text{tag}} \\ \text{form} \end{array} \right\}^*$
 ▷ Evaluate $\text{s} \text{tagbody}$ -like body with *vars* successively bound according to the values of the corresponding *start* and *step* forms. *vars* are bound in parallel/sequentially, respectively. Stop iteration when *stop* is **T**. Return values of *result*. Implicitly, the whole form is a $\text{s} \text{block}$ named **NIL**.

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

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

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

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

$(\text{r} \text{set-dispatch-macro-character} \text{char} \text{sub-char} \text{function} [\text{rt}_{\text{v}*readtable*}])$
 ▷ *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return **T**.

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

13.3 Character Syntax

$\#| \text{multi-line-comment}^* | \#$
 $\#; \text{one-line-comment}^*$

▷ Comments. There are stylistic conventions:

$;;; \text{title}$ ▷ Short title for a block of code.
 $;;; \text{intro}$ ▷ Description before a block of code.
 $:: \text{state}$ ▷ State of program or of following code.
 $;\text{explanation}$ ▷ Regarding line on which it appears.
 $;\text{continuation}$

$(\text{foo}^* [\text{. bar}_{\text{NIL}}])$ ▷ List of *foos* with the terminating *cdr bar*.

$"$ ▷ Begin and end of a string.

$'\text{foo}$ ▷ $(\text{s} \text{quote} \text{foo})$; *foo* unevaluated.

$\`([\text{foo}] [\text{bar}] [,\text{baz}] [,\widetilde{\text{quux}}] [\text{bing}])$
 ▷ Backquote. $\text{s} \text{quote}$ *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.

$\# \backslash c$ ▷ $(\text{r} \text{character} "c")$, the character *c*.

$\# \text{B}n$; $\# \text{O}n$; $n.$; $\# \text{X}n$; $\# \text{rR}n$
 ▷ Integer of radix 2, 8, 10, 16, or *r*; $2 \leq r \leq 36$.

n/d ▷ The **ratio** $\frac{n}{d}$.

$\{ [m].n [\{\text{S|F|D|L|E}\}_x \text{E}_0] | m.[.n] [\{\text{S|F|D|L|E}\}_x] \}$
 ▷ $m.n \cdot 10^x$ as **short-float**, **single-float**, **double-float**, **long-float**, or the type from $\text{*read-default-float-format*}$.

$\# \text{C}(a b)$ ▷ $(\text{r} \text{complex} a b)$, the complex number $a + bi$.

$\#^* \text{foo}$ ▷ $(\text{s} \text{function} \text{foo})$; the function named *foo*.

$\# n \text{A} \text{sequence}$ ▷ *n*-dimensional array.

$\# [n](\text{foo}^*)$
 ▷ Vector of some (or *n*) *foos* filled with last *foo* if necessary.

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

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

$\# \text{P} \text{string}$ ▷ A pathname.

$\# : \text{foo}$ ▷ Uninterned symbol *foo*.

13.2 Reader

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

▷ Ask user a question and return T or NIL depending on their answer. See page 36, *f*format, for *control* and *args*.

(*m*with-standard-io-syntax *form*⁵)

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

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

▷ Read printed representation of object.

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

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

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

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

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

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

▷ Return next character from *stream*.

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

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

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

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

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

▷ Put last *f*read-chared *character* back into *stream*; return NIL.

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

▷ Read next byte from binary *stream*.

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

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

(*f*read-sequence *sequence* *stream* [:start *start* T] [:end *end* NIL]]])

▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.

(*f*readtable-case *readtable*)^{supcase}

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

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

▷ Return copy of *from-readtable*.

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

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

*v**readtable* ▷ Current readtable.

*v**read-base*^T ▷ Radix for reading integers and ratios.

*v**read-default-float-format*^{single-float}

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

(*m*dotimes (*var* *i* [*result* NIL]) (*declare* *decl*^{*})^{*} {*tag*|*form*}^{*})

▷ Evaluate *s*tagbody-like body with *var* successively bound to integers from 0 to *i* – 1. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a *s*block named NIL.

(*m*dolist (*var* *list* [*result* NIL]) (*declare* *decl*^{*})^{*} {*tag*|*form*}^{*})

▷ Evaluate *s*tagbody-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a *s*block named NIL.

9.7 Loop Facility

(*m*loop *form*^{*})

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

(*m*loop *clause*^{*})

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

named *n*NIL ▷ Give *m*loop's implicit *s*block a name.

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

{and $\left\{ \begin{array}{l} \textit{var-p} \\ \textit{var-p}^* \end{array} \right\}$ [*d-type*] [= *bar*]]^{*}

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

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

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

{for | **as** $\left\{ \begin{array}{l} \textit{var-s} \\ \textit{var-s}^* \end{array} \right\}$ [*d-type*]]⁺ **{and** $\left\{ \begin{array}{l} \textit{var-p} \\ \textit{var-p}^* \end{array} \right\}$ [*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* T | *function* *#:cdr*}

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

= *foo* [**then** *bar* foo]

▷ Bind *var* initially to *foo* and later to *bar*.

across *vector*

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

being {**the** | **each**}

▷ Iterate over a hash table or a package.

{hash-key | **hash-keys** } {**of** | **in**} *hash-table* [**using** (*hash-value* *value*)]

▷ Bind *var* successively to the keys of *hash-table*; bind *value* to corresponding values.

{hash-value | **hash-values** } {**of** | **in**} *hash-table* [**using** (*hash-key* *key*)]

▷ Bind *var* successively to the values of *hash-table*; bind *key* to corresponding keys.

{symbol | **symbols** | **present-symbol** | **present-symbols** | **external-symbol** | **external-symbols** } {**of** | **in**} *package* *v**package*

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

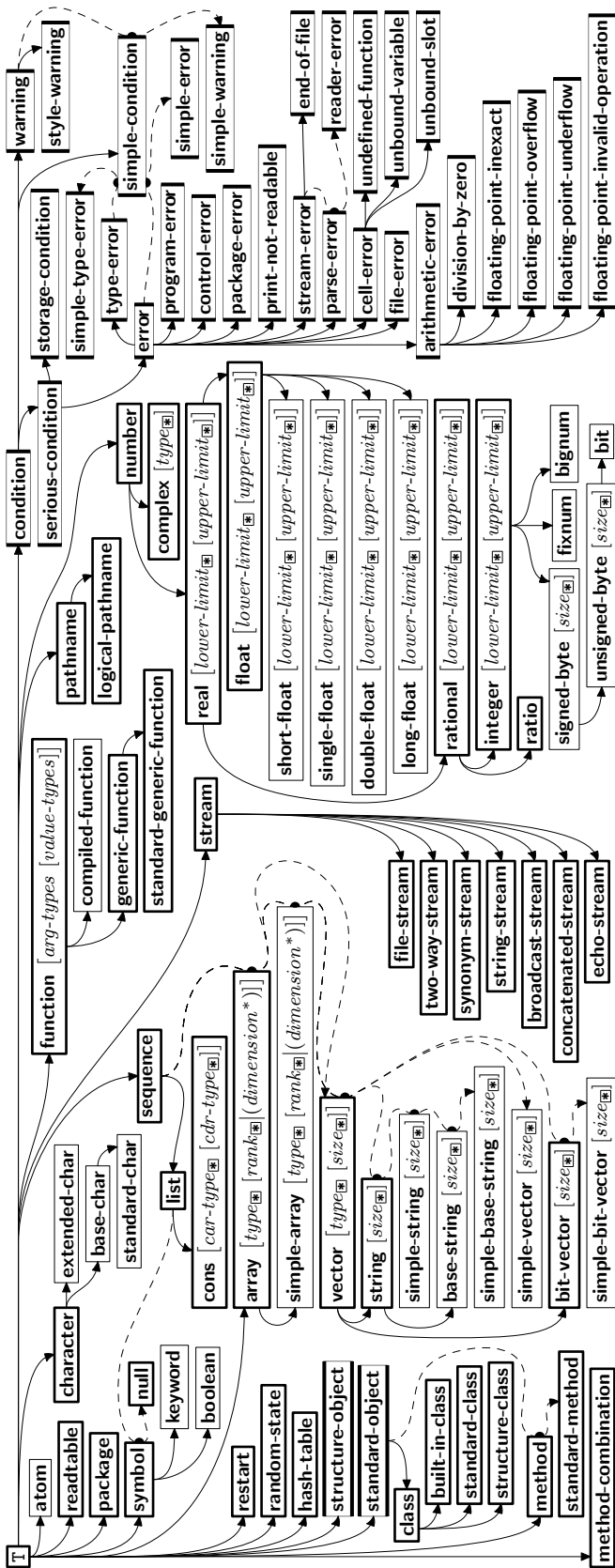


Figure 2: Precedence Order of System Classes (□), Classes (▣), Types (▢), and Condition Types (▧). Every type is also a supertype of NIL, the empty type.

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

10 CLOS

10.1 Classes

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

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

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

(f**find-class** *symbol* [errorp] [environment])
▷ Return class named *symbol*. setfable.

(g**make-instance** *class* initarg *value** other-keyarg*)
▷ Make new instance of *class*.

(g**reinitialize-instance** *instance* initarg *value** other-keyarg*)
▷ Change local slots of instance according to initargs by means of g**shared-initialize**.

(f**slot-value** *foo* *slot*) ▷ Return value of slot in *foo*. setfable.

(f**slot-makunbound** *instance* *slot*)
▷ Make slot in *instance* unbound.

$\left\{ \begin{array}{l} \{ \text{mwith-slots } (\{ \widehat{\text{slot}} (\widehat{\text{var}} \text{ slot}) \}^*) \\ \text{mwith-accessors } (\{ \widehat{\text{var}} \text{ accessor} \}^*) \} \end{array} \right\} \text{ instance } (\text{declare } \widehat{\text{decl}}^*)^* \text{ form}^*$
▷ Return values of forms after evaluating them in a lexical environment with slots of *instance* visible as setfable slots or vars/with accessors of *instance* visible as setfable vars.

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

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

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

(g**make-instances-obsolete** *class*)
▷ Update all existing instances of *class* using g**update-instance-for-redefined-class**.

$\left\{ \begin{array}{l} \text{ginitialize-instance } \text{instance} \\ \text{gupdate-instance-for-different-class } \text{previous } \text{current} \end{array} \right\} \text{ } \{:\text{initarg } \text{value}\}^* \text{ other-keyarg}^*$
▷ Set slots on behalf of g**make-instance**/of g**change-class** by means of g**shared-initialize**.

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

(g**allocate-instance** *class* initarg *value** other-keyarg*)
▷ Return uninitialized instance of *class*. Called by g**make-instance**.

(f**invoke-restart** *restart* *arg**)

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

$\left\{ \begin{array}{l} \text{find-restart} \\ \text{compute-restarts } \text{name} \end{array} \right\} [\text{condition}]$

▷ Return innermost restart *name*, or a list of all restarts, respectively, out of those either associated with *condition* or un-associated at all; or, without *condition*, out of all restarts. Return NIL if search is unsuccessful.

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

$\left\{ \begin{array}{l} \text{abort} \\ \text{muffle-warning} \\ \text{continue} \\ \text{store-value } \text{value} \\ \text{use-value } \text{value} \end{array} \right\} [\text{condition}_{\text{NIL}}]$

▷ 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 f**abort** and f**muffle-warning**, or return NIL for the rest.

(m**with-condition-restarts** *condition* *restarts* *form**)
▷ Evaluate *forms* with *restarts* dynamically associated with *condition*. Return values of *forms*.

(f**arithmetic-error-operation** *condition*)

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

(f**cell-error-name** *condition*)

▷ Name of cell which caused *condition*.

(f**unbound-slot-instance** *condition*)

▷ Instance with unbound slot which caused *condition*.

(f**print-not-readable-object** *condition*)

▷ The object not readably printable under *condition*.

(f**package-error-package** *condition*)

(f**file-error-pathname** *condition*)

(f**stream-error-stream** *condition*)

▷ Package, path, or stream, respectively, which caused the *condition* of indicated type.

(f**type-error-datum** *condition*)

(f**type-error-expected-type** *condition*)

▷ Object which caused *condition* of type **type-error**, or its expected type, respectively.

(f**simple-condition-format-control** *condition*)

(f**simple-condition-format-arguments** *condition*)

▷ Return f**format** control or list of f**format** arguments, respectively, of *condition*.

*break-on-signals*NIL

▷ Condition type debugger is to be invoked on.

*debugger-hook*NIL

▷ Function of condition and function itself. Called before debugger.

12 Types and Classes

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

(f**typep** *foo* *type* [environment]) ▷ T if *foo* is of *type*.

(f**subtypep** *type-a* *type-b* [environment])

▷ Return T if *type-a* is a recognizable subtype of *type-b*, and NIL if the relationship could not be determined.

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

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

(*m*ignore-errors *form*^P)

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

(*f*invoke-debugger *condition*)

▷ Invoke debugger with *condition*.

(*m*assert *test* [(*place*^{*}) $\left\{ \begin{array}{l} \text{condition } \textit{continue-arg}^* \\ \text{condition-type } \{:\textit{initarg-name value}\}^* \\ \text{control } \textit{arg}^* \end{array} \right\}$]])

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

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

▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-forms* with *var* bound to the condition, and return their values. Without a condition, bind *ord-λs* to values of *foo* and return values of forms or, without a **no-error** clause, return values of foo. See page 17 for (*ord-λ*^{*}).

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

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

(*m*with-simple-restart ($\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\}$ *control* *arg*^{*}) *form*^P)

▷ Return values of forms unless *restart* is called during their evaluation. In this case, describe *restart* using *f*format *control* and *args* (see page 36) and return NIL and T.

(*m*restart-case *form* (*restart* (*ord-λ*^{*}) $\left\{ \begin{array}{l} \text{:interactive } \textit{arg-function} \\ \text{:report } \left\{ \begin{array}{l} \textit{report-function} \\ \text{string } \textit{restart}^* \end{array} \right\} \\ \text{:test } \textit{test-function} \square \end{array} \right\}$)

(declare $\widehat{\text{decl}}^*$)^{*} *restart-form*^P)^{*}
▷ Return values of form or, if during evaluation of *form* one of the dynamically established *restarts* is called, the values of its restart-forms. A *restart* is visible under *condition* if (*funcall* *#'test-function* *condition*) returns T. If presented in the debugger, *restarts* are described by *string* or by *#'report-function* (of a stream). A *restart* can be called by (*invoke-restart* *restart* *arg*^{*}), where *args* match *ord-λ*^{*}, or by (*invoke-restart-interactively* *restart*) where a list of the respective *args* is supplied by *#'arg-function*. See page 17 for *ord-λ*^{*}.

(*m*restart-bind (($\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\}$ *restart-function*

$\left\{ \begin{array}{l} \text{:interactive-function } \textit{arg-function} \\ \text{:report-function } \textit{report-function} \\ \text{:test-function } \textit{test-function} \end{array} \right\}$)^{*}) *form*^P)

▷ Return values of forms evaluated with dynamically established *restarts* whose *restart-functions* should perform a non-local transfer of control. A restart is visible under *condition* if (*test-function* *condition*) returns T. If presented in the debugger, *restarts* are described by *restart-function* (of a stream). A *restart* can be called by (*invoke-restart* *restart* *arg*^{*}), where *args* must be suitable for the corresponding *restart-function*, or by (*invoke-restart-interactively* *restart*) where a list of the respective *args* is supplied by *arg-function*.

(*g*shared-initialize *instance* $\left\{ \begin{array}{l} \textit{initform-slots} \\ \text{T} \end{array} \right\} \{:\textit{initarg-slot value}\}^*$

other-keyarg^{*})

▷ Fill the *initarg-slots* of *instance* with the corresponding *values*, and fill those *initform-slots* that are not *initarg-slots* with the values of their *initform* forms.

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

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

▷ Called on attempted access to non-existing or unbound *slot*. Default methods signal **error/unbound-slot**, respectively. Not to be called by user.

10.2 Generic Functions

(*f*next-method-p) ▷ T if enclosing method has a next method.

(*m*defgeneric $\left\{ \begin{array}{l} \textit{foo} \\ (\text{setf } \textit{foo}) \end{array} \right\}$ (*required-var*^{*} [*&optional* $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var}) \end{array} \right\}^*$]

[*&rest* *var*] [*&key* $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [:\textit{key } \textit{var}]) \end{array} \right\}^*$] [*&allow-other-keys*])

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

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

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

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

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

(*m*defmethod $\left\{ \begin{array}{l} \textit{foo} \\ (\text{setf } \textit{foo}) \end{array} \right\}$ $\left\{ \begin{array}{l} \text{:before} \\ \text{:after} \\ \text{:around} \end{array} \right\} \left\{ \begin{array}{l} \textit{primary method} \\ \textit{qualifier}^* \end{array} \right\}$)

$\left\{ \begin{array}{l} \textit{var} \\ (\textit{spec-var } \left\{ \begin{array}{l} \textit{class} \\ (\text{eql } \textit{bar}) \end{array} \right\}) \end{array} \right\}^*$ [*&optional* $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [\textit{init} [\textit{supplied-p}]]]) \end{array} \right\}^*$] [*&rest* *var*] [*&key* $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [\textit{key } \textit{var}]) \end{array} \right\}^*$] [*&allow-other-keys*]

[*&aux* $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [\textit{init}]) \end{array} \right\}^*$] $\left\{ \begin{array}{l} \text{(declare } \widehat{\text{decl}}^* \text{)}^* \\ \textit{doc} \end{array} \right\}$ *form*^P)

▷ Define new method for generic function *foo*. *spec-vars* specialize to either being of *class* or being *eql* *bar*, respectively. On invocation, *vars* and *spec-vars* of the new method act like parameters of a function with body *form*^{*}. *forms* are enclosed in an implicit *block* *foo*. Applicable *qualifiers* depend on the *method-combination* type; see section 10.3.

