

Quick Reference

lisp

Common

lisp



Contents

1	Numbers	3	9.5	Control Flow . . .	21
1.1	Predicates	3	9.6	Iteration	22
1.2	Numeric Functns .	3	9.7	Loop Facility . . .	22
1.3	Logic Functions .	5	10	CLOS	25
1.4	Integer Functions .	6	10.1	Classes	25
1.5	Implementation-Dependent	6	10.2	Generic Functns .	26
2	Characters	7	10.3	Method Combination Types . . .	28
3	Strings	8	11	Conditions and Errors	28
4	Conses	8	12	Types and Classes	31
4.1	Predicates	8	13	Input/Output	33
4.2	Lists	9	13.1	Predicates	33
4.3	Association Lists .	10	13.2	Reader	33
4.4	Trees	10	13.3	Character Syntax .	35
4.5	Sets	11	13.4	Printer	36
5	Arrays	11	13.5	Format	38
5.1	Predicates	11	13.6	Streams	40
5.2	Array Functions .	11	13.7	Paths and Files . .	42
5.3	Vector Functions .	12	14	Packages and Symbols	43
6	Sequences	12	14.1	Predicates	43
6.1	Seq. Predicates . .	12	14.2	Packages	44
6.2	Seq. Functions . .	13	14.3	Symbols	45
7	Hash Tables	15	14.4	Std Packages . . .	46
8	Structures	16	15	Compiler	46
9	Control Structure	16	15.1	Predicates	46
9.1	Predicates	16	15.2	Compilation	46
9.2	Variables	17	15.3	REPL & Debug . .	47
9.3	Functions	18	15.4	Declarations . . .	48
9.4	Macros	19	16	External Environment	49

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.
<i>me</i>	▷ Literal text.
[<i>foo</i> <u><i>bar</i></u>]	▷ Either one <i>foo</i> or nothing; defaults to <i>bar</i> .
<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> }; $\begin{cases} foo \\ bar \\ baz \end{cases}$	▷ Either <i>foo</i> , or <i>bar</i> , or <i>baz</i> .
$\begin{cases} foo \\ bar \\ baz \end{cases}$	▷ Anything from none to each of <i>foo</i> , <i>bar</i> , and <i>baz</i> .
\widehat{foo}	▷ Argument <i>foo</i> is not evaluated.
\widetilde{bar}	▷ Argument <i>bar</i> is possibly modified.
<i>foo</i> ^P *	▷ <i>foo</i> * is evaluated as in <i>sprogn</i> ; see page 21.
<u><i>foo</i></u> ; <u><i>bar</i></u> ; <u><i>baz</i></u> _{<i>n</i>}	▷ Primary, secondary, and <i>n</i> th return value.
T; NIL	▷ t , or truth in general; and nil or () .

1 Numbers

1.1 Predicates

- $(f = number^+)$
 $(f \neq number^+)$ \triangleright \underline{T} if all *numbers*, or none, respectively, are equal in value.
- $(f > number^+)$
 $(f \geq number^+)$
 $(f < number^+)$
 $(f \leq number^+)$ \triangleright Return \underline{T} if *numbers* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.
- $(f \text{minusp } a)$
 $(f \text{zerop } a)$ \triangleright \underline{T} if $a < 0$, $a = 0$, or $a > 0$, respectively.
 $(f \text{plusp } a)$
- $(f \text{evenp } int)$ \triangleright \underline{T} if *int* is even or odd, respectively.
 $(f \text{oddp } int)$
- $(f \text{numberp } foo)$
 $(f \text{realp } foo)$
 $(f \text{rationalp } foo)$
 $(f \text{floatp } foo)$ \triangleright \underline{T} if *foo* is of indicated type.
 $(f \text{integerp } foo)$
 $(f \text{complexp } foo)$
 $(f \text{random-state-p } foo)$

1.2 Numeric Functions

- $(f + a_{\square}^*)$ \triangleright Return $\sum a$ or $\prod a$, respectively.
 $(f * a_{\square}^*)$
- $(f - a b^*)$
 $(f / a b^*)$ \triangleright Return $a - \sum b$ or $a / \prod b$, respectively. Without any *bs*, return $-a$ or $1/a$, respectively.
- $(f 1+ a)$ \triangleright Return $a + 1$ or $a - 1$, respectively.
 $(f 1- a)$
- $\left\{ \begin{matrix} m \text{incf} \\ m \text{decf} \end{matrix} \right\} \widetilde{place} [delta_{\square}]$
 \triangleright Increment or decrement the value of *place* by *delta*. Return new value.
- $(f \text{exp } p)$ \triangleright Return e^p or b^p , respectively.
 $(f \text{expt } b p)$
- $(f \text{log } a [b_{\square}])$ \triangleright Return $\log_b a$ or, without *b*, $\ln a$.
- $(f \text{sqrt } n)$ \triangleright \sqrt{n} in complex numbers/natural numbers.
 $(f \text{isqrt } n)$
- $(f \text{lcm } integer^*_{\square})$
 $(f \text{gcd } integer^*)$ \triangleright Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns $\underline{0}$.
- cpi \triangleright **long-float** approximation of π , Ludolph's number.
- $(f \text{sin } a)$
 $(f \text{cos } a)$ \triangleright $\sin a$, $\cos a$, or $\tan a$, respectively. (*a* in radians.)
 $(f \text{tan } a)$
- $(f \text{asin } a)$ \triangleright $\arcsin a$ or $\arccos a$, respectively, in radians.
 $(f \text{acos } a)$
- $(f \text{atan } a [b_{\square}])$ \triangleright $\arctan \frac{a}{b}$ in radians.

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

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

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

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

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

$\left\{ \begin{array}{l} \{f\text{round} \mid f\text{round}\} \\ \{f\text{floor} \mid f\text{ffloor}\} \\ \{f\text{ceiling} \mid f\text{fceil}\} \\ \{f\text{truncate} \mid f\text{ftruncate}\} \end{array} \right\} n \ [d_{\square}]$
 ▷ 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* [*state* *v**random-state*])
 ▷ Return non-negative random number less than *limit*, and of the same type.

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

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

(*f***float-sign** *num-a* [*num-b* _{\square}]) ▷ *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* _{\square}]) ▷ Make a complex number.

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

(*f***abs** *n*) ▷ Return $|n|$.

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

(*f***float** *real* [*prototype* 0.0f0])
 ▷ Convert *real* into float with type of *prototype*.

USER-HOMEDIR- PATHNAME 42	WARN 29 WARNING 32 WHEN 21, 24 WHILE 24 WILD-PATHNAME-P 33	FROM-STRING 41 WITH-OPEN-FILE 41 WITH-OPEN-STREAM 41 WITH-OUTPUT- TO-STRING 42 WITH-ACCESSORS 25 WITH-PACKAGE- ITERATOR 45 WITH-SIMPLE- RESTART 30 WITH-SLOTS 25 WITH-STANDARD- IO-SYNTAX 33 WRITE 37	WRITE-BYTE 36 WRITE-CHAR 36 WRITE-LINE 36 WRITE-SEQUENCE 36 WRITE-STRING 36 WRITE-TO-STRING 37
USING 24	V 40 VALUES 18, 33 VALUES-LIST 18 VARIABLE 45 VECTOR 12, 32 VECTOR-POP 12 VECTOR-PUSH 12 VECTOR- PUSH-EXTEND 12 VECTORP 11	WITH-CONDITION- RESTARTS 30 WITH-HASH-TABLE- ITERATOR 15 WITH-INPUT-	Y-OR-N-P 33 YES-OR-NO-P 33 ZEROP 3

NAME-CHAR 7
 NAMED 22
 NAMESTRING 43
 NBUTLAST 9
 NCONC 10, 24, 28
 NCONCING 24
 NEVER 25
 NEWLINE 7
 NEXT-METHOD-P 26
 NIL 2, 46
 NINTERSECTION 11
 NINTH 9
 NO-APPLICABLE-METHOD 27
 NO-NEXT-METHOD 27
 NOT 16, 33, 36
 NOTANY 12
 NOTEVERY 12
 NOTINLINE 48
 NRECONC 10
 NREVERSE 13
 NSET-DIFFERENCE 11
 NSET-EXCLUSIVE-OR 11
 NSTRING-CAPITALIZE 8
 NSTRING-DOWNCASE 8
 NSTRING-UPCASE 8
 NSUBLIS 11
 NSUBST 10
 NSUBST-IF 10
 NSUBST-IF-NOT 10
 NSUBSTITUTE 14
 NSUBSTITUTE-IF 14
 NSUBSTITUTE-IF-NOT 14
 NTH 9
 NTH-VALUE 18
 NTHCDR 9
 NULL 8, 32
 NUMBER 32
 NUMBERP 3
 NUMERATOR 4
 NUNION 11
 ODDP 3
 OF 24
 OF-TYPE 22
 ON 22
 OPEN 40
 OPEN-STREAM-P 33
 OPTIMIZE 49
 OR 21, 28, 33, 36
 OTHERWISE 21, 31
 OUTPUT-STREAM-P 33
 PACKAGE 32
 PACKAGE-ERROR 32
 PACKAGE-ERROR-P 31
 PACKAGE-NAME 44
 PACKAGE-NICKNAMES 44
 PACKAGE-SHADOWING-SYMBOLS 45
 PACKAGE-USE-LIST 44
 PACKAGE-USED-BY-LIST 44
 PACKAGEP 43
 PAIRLIS 10
 PARSE-ERROR 32
 PARSE-INTEGER 8
 PARSE-NAMESTRING 42
 PATHNAME 32, 43
 PATHNAME-DEVICE 42
 PATHNAME-DIRECTORY 42
 PATHNAME-HOST 42
 PATHNAME-MATCH-P 33
 PATHNAME-NAME 42
 PATHNAME-TYPE 42
 PATHNAME-VERSION 42
 PATHNAMEP 33
 PEEK-CHAR 34
 PHASE 4
 PI 3
 PLUSP 3
 POP 9
 POSITION 13
 POSITION-IF 14
 POSITION-IF-NOT 14
 PPRINT 36
 PPRINT-DISPATCH 38
 PPRINT-EXIT-IF-LIST-EXHAUSTED 37
 PPRINT-FILL 37
 PPRINT-INDENT 37
 PPRINT-LINEAR 37
 PPRINT-LOGICAL-BLOCK 37
 PPRINT-NEWLINE 37
 PPRINT-POP 37
 PPRINT-TAB 37
 PPRINT-TABULAR 37
 PRESENT-SYMBOL 24
 PRESENT-SYMBOLS 24
 PRIN1 36
 PRIN1-TO-STRING 36
 PRINC 36
 PRINC-TO-STRING 36
 PRINT 36
 PRINT-
 NOT-READABLE 32
 PRINT-NOT-
 READABLE-OBJECT 31
 PRINT-OBJECT 36
 PRINT-UNREADABLE-OBJECT 36
 PROBE-FILE 43
 PROCLAIM 48
 PROG 21
 PROG1 21
 PROG2 21
 PROG* 21
 PROGN 21, 28
 PROGRAM-ERROR 32
 PROGV 17
 PROVIDE 45
 PSETF 17
 PSETQ 17
 PUSH 10
 PUSHNEW 10
 QUOTE 35, 47
 RANDOM 4
 RANDOM-STATE 32
 RANDOM-STATE-P 3
 RASSOC 10
 RASSOC-IF 10
 RASSOC-IF-NOT 10
 RATIO 32, 35
 RATIONAL 4, 32
 RATIONALIZE 4
 RATIONALP 3
 READ 33
 READ-BYTE 34
 READ-CHAR 34
 READ-CHAR-NO-HANG 34
 READ-
 DELIMITED-LIST 34
 READ-FROM-STRING 33
 READ-LINE 34
 READ-PRESERVING-WHITESPACE 33
 READ-SEQUENCE 34
 READER-ERROR 32
 READTABLE 32
 READTABLE-CASE 34
 READTABLEP 33
 REAL 32
 REALP 3
 REALPART 4
 REDUCE 15
 REINITIALIZE-INSTANCE 25
 REM 4
 REMF 17
 REMHASH 15
 REMOVE 14
 REMOVE-DUPPLICATES 14
 REMOVE-IF 14
 REMOVE-IF-NOT 14
 REMOVE-METHOD 27
 REMPROP 17
 RENAME-FILE 43
 RENAME-PACKAGE 44
 REPEAT 24
 REPLACE 14
 REQUIRE 45
 REST 9
 RESTART 32
 RESTART-BIND 30
 RESTART-CASE 30
 RESTART-NAME 30
 RETURN 21, 24
 RETURN-FROM 21
 REVAPPEND 10
 REVERSE 13
 ROOM 48
 ROTATEF 17
 ROUND 4
 ROW-MAJOR-AREF 11
 RPLACA 9
 RPLACD 9
 SAFETY 49
 SATISFIES 33
 SBIT 12
 SCALE-FLOAT 6
 SCHAR 8
 SEARCH 14
 SECOND 9
 SEQUENCE 32
 SERIOUS-CONDITION 32
 SET 17
 SET-DIFFERENCE 11
 SET-
 DISPATCH-MACRO-CHARACTER 35
 SET-EXCLUSIVE-OR 11
 SET-MACRO-CHARACTER 34
 SET-PPRINT-DISPATCH 38
 SET-SYNTAX-FROM-CHAR 34
 SETF 17, 45
 SETQ 17
 SEVENTH 9
 SHADOW 44
 SHADOWING-IMPORT 44
 SHARED-INITIALIZE 26
 SHIFTF 17
 SHORT-FLOAT 32, 35
 SHORT-
 FLOAT-EPSILON 6
 SHORT-FLOAT-NEGATIVE-EPSILON 6
 SHORT-SITE-NAME 49
 SIGNAL 29
 SIGNED-BYTE 32
 SIGNUM 4
 SIMPLE-ARRAY 32
 SIMPLE-BASE-STRING 32
 SIMPLE-BIT-VECTOR 32
 SIMPLE-
 BIT-VECTOR-P 11
 SIMPLE-CONDITION 32
 SIMPLE-CONDITION-FORMAT-
 ARGUMENTS 31
 SIMPLE-CONDITION-FORMAT-CONTROL 31
 SIMPLE-ERROR 32
 SIMPLE-STRING 32
 SIMPLE-STRING-P 8
 SIMPLE-TYPE-ERROR 32
 SIMPLE-VECTOR 32
 SIMPLE-VECTOR-P 11
 SIMPLE-WARNING 32
 SIN 3
 SINGLE-FLOAT 32, 35
 SINGLE-
 FLOAT-EPSILON 6
 SINGLE-FLOAT-NEGATIVE-EPSILON 6
 SINH 4
 SIXTH 9
 SLEEP 22
 SLOT-BOUND 25
 SLOT-EXISTS-P 25
 SLOT-MAKUNBOUND 25
 SLOT-MISSING 26
 SLOT-UNBOUND 26
 SLOT-VALUE 25
 SOFTWARE-TYPE 49
 SOFTWARE-VERSION 49
 SOME 12
 SORT 13
 SPACE 7, 49
 SPECIAL 49
 SPECIAL-OPERATOR-P 46
 SPEED 49
 SQRT 3
 STABLE-SORT 13
 STANDARD 28
 STANDARD-CHAR 7, 32
 STANDARD-CHAR-P 7
 STANDARD-CLASS 32
 STANDARD-GENERIC-FUNCTION 32
 STANDARD-METHOD 32
 STANDARD-OBJECT 32
 STEP 48
 STORAGE-CONDITION 32
 STORE-VALUE 30
 STREAM 32
 STREAM-
 ELEMENT-TYPE 31
 STREAM-ERROR 32
 STREAM-
 ERROR-STREAM 31
 STREAM-EXTERNAL-FORMAT 42
 STREAMP 33
 STRING 8, 32
 STRING-CAPITALIZE 8
 STRING-DOWNCASE 8
 STRING-EQUAL 8
 STRING-GREATERP 8
 STRING-LEFT-TRIM 8
 STRING-LESSP 8
 STRING-NOT-EQUAL 8
 STRING-
 NOT-GREATERP 8
 STRING-NOT-LESSP 8
 STRING-RIGHT-TRIM 8
 STRING-STREAM 32
 STRING-TRIM 8
 STRING-UPCASE 8
 STRING/= 8
 STRING< 8
 STRING<= 8
 STRING= 8
 STRING> 8
 STRING>= 8
 STRINGP 8
 STRUCTURE 45
 STRUCTURE-CLASS 32
 STRUCTURE-OBJECT 32
 STYLE-WARNING 32
 SUBLIS 11
 SUBSEQ 13
 SUBSETP 9
 SUBST 10
 SUBST-IF 10
 SUBST-IF-NOT 10
 SUBSTITUTE 14
 SUBSTITUTE-IF 14
 SUBSTITUTE-IF-NOT 14
 SUBTYPEP 31
 SUM 24
 SUMMING 24
 SVREF 12
 SXHASH 15
 SYMBOL 24, 32, 45
 SYMBOL-FUNCTION 45
 SYMBOL-MACROLET 20
 SYMBOL-NAME 45
 SYMBOL-PACKAGE 45
 SYMBOL-PLIST 45
 SYMBOL-VALUE 45
 SYMBOLP 43
 SYMBOLS 24
 SYNONYM-STREAM 32
 SYNONYM-STREAM-SYMBOL 41
 T 2, 32, 46
 TAGBODY 21
 TAILP 9
 TAN 3
 TANH 4
 TENTH 9
 TERPRI 36
 THE 24, 31
 THEN 24
 THEREIS 25
 THIRD 9
 THROW 22
 TIME 48
 TO 22
 TRACE 48
 TRANSLATE-LOGICAL-PATHNAME 43
 TRANSLATE-PATHNAME 43
 TREE-EQUAL 10
 TRUENAME 43
 TRUNCATE 4
 TWO-WAY-STREAM 32
 TWO-WAY-STREAM-INPUT-STREAM 41
 TWO-WAY-STREAM-OUTPUT-STREAM 41
 TYPE 45, 48
 TYPE-ERROR 32
 TYPE-ERROR-DATUM 31
 TYPE-ERROR-EXPECTED-TYPE 31
 TYPE-OF 31
 TYPECASE 31
 TYPEP 31
 UNBOUND-SLOT 32
 UNBOUND-SLOT-INSTANCE 31
 UNBOUND-VARIABLE 32
 UNDEFINED-FUNCTION 32
 UNEXPORT 45
 UNINTERN 44
 UNION 11
 UNLESS 21, 24
 UNREAD-CHAR 34
 UNSIGNED-BYTE 32
 UNTIL 24
 UNTRACE 48
 UNUSE-PACKAGE 44
 UNWIND-PROTECT 21
 UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 26
 UPDATE-INSTANCE-FOR-REDEFINED-CLASS 26
 UPFROM 22
 UPGRADED-ARRAY-ELEMENT-TYPE 31
 UPGRADED-COMPLEX-PART-TYPE 6
 UPPER-CASE-P 7
 UPTO 22
 USE-PACKAGE 44
 USE-VALUE 30

1.3 Logic Functions

Negative integers are used in two's complement representation.

(*f* **boole** *operation int-a int-b*)

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

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

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

(*f* **logeqv** *integer**)

(*f* **logand** *integer**)

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

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

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

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

(*f* **logxor** *integer**)

(*f* **logior** *integer**)

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

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

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

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

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

(*f* **logtest** *int-a int-b*)

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

(*f* **logcount** *int*)

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

Index

```

" 35
' 35
( 35
) 46
) 35
• 3, 32, 33, 43, 47
•• 43, 47
••• 47
•BREAK-
  ON-SIGNALS* 31
•COMPILE-FILE-
  PATHNAME* 46
•COMPILE-FILE-
  TRUENAME* 46
•COMPILE-PRINT* 46
•COMPILE-VERBOSE*
  46
•DEBUG-IO* 42
•DEBUGGER-HOOK*
  31
•DEFAULT-
  PATHNAME-
  DEFAULTS* 42
•ERROR-OUTPUT* 42
•FEATURES* 36
•GENSYM-COUNTER*
  45
•LOAD-PATHNAME*
  46
•LOAD-PRINT* 46
•LOAD-TRUENAME*
  46
•LOAD-VERBOSE* 46
•MACROEXPAND-
  HOOK* 48
•MODULES* 45
•PACKAGE* 44
•PRINT-ARRAY* 37
•PRINT-BASE* 37
•PRINT-CASE* 38
•PRINT-CIRCLE* 38
•PRINT-ESCAPE* 38
•PRINT-GENSYM* 38
•PRINT-LENGTH* 38
•PRINT-LEVEL* 38
•PRINT-LINES* 38
•PRINT-
  MISER-WIDTH* 38
•PRINT-PPRINT-
  DISPATCH* 38
•PRINT-PRETTY* 38
•PRINT-RADIX* 38
•PRINT-READABLY*
  38
•PRINT-
  RIGHT-MARGIN* 38
•QUERY-IO* 42
•RANDOM-STATE* 4
•READ-BASE* 34
•READ-DEFAULT-
  FLOAT-FORMAT* 34
•READ-EVAL* 35
•READ-SUPPRESS* 34
•READTABLE* 34
•STANDARD-INPUT*
  42
•STANDARD-
  OUTPUT* 42
•TERMINAL-IO* 42
•TRACE-OUTPUT* 48
+ 3, 28, 47
++ 47
+++ 47
. 35
,@ 35
- 3, 47
. 35
/ 3, 35, 47
// 47
/// 47
/= 3
: 44
:: 44
:ALLOW-OTHER-KEYS
  20
: 35
< 3
<= 3
= 3, 22, 24
> 3
>= 3
\ 36
# 40
#\ 35
#' 35
#( 35
#* 35
#+ 36
#- 36
# 35
#: 35
#< 35
#= 35
#A 35
#B 35
#C( 35
#O 35
#P 35
#R 35
#S( 35
#X 35
## 35
#| 35
&ALLOW-
  OTHER-KEYS 20
&AUX 20
&BODY 20
&ENVIRONMENT 20
&KEY 20
&OPTIONAL 20
&REST 20
&WHOLE 20
~( ~) 39
~* 40
~/ 40
~< ~> 39
~< ~> 39
~? 40
~A 38
~B 39
~C 39
~D 39
~E 39
~F 39
~G 39
~I 40
~O 39
~P 39
~R 38
~S 38
~T 40
~W 40
~X 39
~[ ~] 40
~$ 39
~% 39
~& 39
~' 40
~. 39
~| 39
~{ ~} 40
~< 39
~< 39
~ 35
| 36
1+ 3
1- 3
ABORT 30
ABOVE 22
ABS 4
ACONS 10
ACOS 3
ACOSH 4
ACROSS 24
ADD-METHOD 27
ADJOIN 9
ADJUST-ARRAY 11
ADJUSTABLE-
  ARRAY-P 11
ALLOCATE-INSTANCE
  26
ALPHA-CHAR-P 7
ALPHANUMERICP 7
ALWAYS 25
AND
  21, 22, 24, 28, 33, 36
APPEND 10, 24, 28
APPENDING 24
APPLY 18
APPROPOS 47
APPROPOS-LIST 47
AREF 11
ARITHMETIC-ERROR
  32
ARITHMETIC-ERROR-
  OPERANDS 30
ARITHMETIC-ERROR-
  OPERATION 30
ARRAY 32
ARRAY-DIMENSION 11
ARRAY-DIMENSION-
  LIMIT 12
ARRAY-DIMENSIONS
  11
ARRAY-
  DISPLACEMENT 11
ARRAY-
  ELEMENT-TYPE 31
ARRAY-HAS-
  FILL-POINTER-P 11
ARRAY-IN-BOUNDS-P
  11
ARRAY-RANK 11
ARRAY-RANK-LIMIT 12
ARRAY-ROW-
  MAJOR-INDEX 11
ARRAY-TOTAL-SIZE 11
ARRAY-TOTAL-
  SIZE-LIMIT 12
ARRAYP 11
AS 22
ASH 6
ASIN 3
ASINH 4
ASSERT 29
ASSOC 10
ASSOC-IF 10
ASSOC-IF-NOT 10
ATAN 3
ATANH 4
ATOM 9, 32
BASE-CHAR 32
BASE-STRING 32
BEING 24
BELOW 22
BIGNUM 32
BIT 12, 32
BIT-AND 12
BIT-ANDC1 12
BIT-ANDC2 12
BIT-EQV 12
BIT-IOR 12
BIT-NAND 12
BIT-NOR 12
BIT-NOT 12
BIT-ORC1 12
BIT-ORC2 12
BIT-VECTOR 32
BIT-VECTOR-P 11
BIT-XOR 12
BLOCK 21
BOOLE 5
BOOLE-1 5
BOOLE-2 5
BOOLE-AND 5
BOOLE-ANDC1 5
BOOLE-ANDC2 5
BOOLE-C1 5
BOOLE-C2 5
BOOLE-CLR 5
BOOLE-EQV 5
BOOLE-IOR 5
BOOLE-NAND 5
BOOLE-NOR 5
BOOLE-ORC1 5
BOOLE-ORC2 5
BOOLE-SET 5
BOOLE-XOR 5
BOOLEAN 32
BOTH-CASE-P 7
BOUNDP 16
BREAK 48
BROADCAST-STREAM
  32
BROADCAST-
  STREAM-STREAMS
  41
BUILT-IN-CLASS 32
BUTLAST 9
BY 22
BYTE 6
BYTE-POSITION 6
BYTE-SIZE 6
CAAR 9
CADR 9
CALL-ARGUMENTS-
  LIMIT 19
CALL-METHOD 28
CALL-NEXT-METHOD
  27
CAR 9
CASE 21
CATCH 22
CCASE 21
CDAR 9
CDDR 9
CDR 9
CEILING 4
CELL-ERROR 32
CELL-ERROR-NAME 31
CERROR 29
CHANGE-CLASS 26
CHAR 8
CHAR-CODE 7
CHAR-CODE-LIMIT 7
CHAR-DOWNCASE 7
CHAR-EQUAL 7
CHAR-GREATERP 7
CHAR-INT 7
CHAR-LESSP 7
CHAR-NAME 7
CHAR-NOT-EQUAL 7
CHAR-NOT-GREATERP
  7
CHAR-NOT-LESSP 7
CHAR-UPCASE 7
CHAR/= 7
CHAR< 7
CHAR<= 7
CHAR= 7
CHAR> 7
CHAR>= 7
CHARACTER 7, 32, 35
CHARACTERP 7
CHECK-TYPE 31
CIS 4
CL 46
CL-USER 46
CLASS 32
CLASS-NAME 25
CLASS-OF 25
CLEAR-INPUT 41
CLEAR-OUTPUT 41
CLOSE 41
CLQR 1
CLRHASH 15
CODE-CHAR 7
COERCE 31
COLLECT 24
COLLECTING 24
COMMON-LISP 46
COMMON-LISP-USER
  46
COMPILATION-SPEED
  49
COMPILE 46
COMPILE-FILE 46
COMPILE-
  FILE-PATHNAME 46
COMPILED-FUNCTION
  32
COMPILED-
  FUNCTION-P 46
COMPILER-MACRO 45
COMPILER-MACRO-
  FUNCTION 47
COMPLEMENT 18
COMPLEX 4, 32, 35
COMPLEXP 3
COMPUTE
  APPLICABLE-
  METHODS 27
COMPUTE-RESTARTS
  30
CONCATENATE 13
CONCATENATED-
  STREAM 32
CONCATENATED-
  STREAM-STREAMS
  41
COND 21
CONDITION 32
CONJUGATE 4
CONS 9, 32
CONSP 8
CONSTANTLY 19
CONSTANTP 17
CONTINUE 30
CONTROL-ERROR 32
COPY-ALIST 10
COPY-LIST 10
COPY-PPRINT-
  DISPATCH 38
COPY-READTABLE 34
COPY-SEQ 15
COPY-STRUCTURE 16
COPY-SYMBOL 45
COPY-TREE 11
COS 3
COSH 4
COUNT 13, 24
COUNT-IF 13
COUNT-IF-NOT 13
COUNTING 24
CTYPECASE 31
DEBUG 49
DECF 3
DECLAIM 48
DECLARATION 48
DECLARE 48
DECODE-FLOAT 6
DECODE-UNIVERSAL-
  TIME 49
DEFCLASS 25
DEFCONSTANT 17
DEFGeneric 26
DEFINE-COMPILER-
  MACRO 19
DEFINE-CONDITION 29
DEFINE-METHOD-
  COMBINATION 28
DEFINE-
  MODIFY-MACRO 20
DEFINE-
  SETF-EXPANDER 20
DEFINE-
  SYMBOL-MACRO 19
DEFMACRO 19
DEFMETHOD 27
DEFPACKAGE 44
DEFPARAMETER 17
DEFSETF 20
DESTRUCT 16
DEFTYPE 33
DEFUN 18
DEFVAR 17
DELETE 14
DELETE-DUPLICATES
  14
DELETE-FILE 43
DELETE-IF 14
DELETE-IF-NOT 14
DELETE-PACKAGE 44
DENOMINATOR 4
DEPOSIT-FIELD 6
DESCRIBE 48
DESCRIBE-OBJECT 48
DESTRUCTURING-
  BIND 18
DIGIT-CHAR 7

```

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*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*₁₀]) ▷ Character representing digit *i*.

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

(*f*name-char *foo*) ▷ Character named *foo* if any, or NIL.

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

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

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

(*f*character *c*) ▷ Return #\c.

3 Strings

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

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

$\left\{ \begin{array}{l} \text{fstring=} \\ \text{fstring-equal} \end{array} \right\} \text{foo bar} \left\{ \begin{array}{l} \text{:start1 start-foo}_{\text{0}} \\ \text{:start2 start-bar}_{\text{0}} \\ \text{:end1 end-foo}_{\text{NIL}} \\ \text{:end2 end-bar}_{\text{NIL}} \end{array} \right\}$
▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.

$\left\{ \begin{array}{l} \text{fstring}{/=|-not-equal} \\ \text{fstring}{>|-greaterp} \\ \text{fstring}{>=|-not-lessp} \\ \text{fstring}{<|-lessp} \\ \text{fstring}{<=|-not-greaterp} \end{array} \right\} \text{foo bar} \left\{ \begin{array}{l} \text{:start1 start-foo}_{\text{0}} \\ \text{:start2 start-bar}_{\text{0}} \\ \text{:end1 end-foo}_{\text{NIL}} \\ \text{:end2 end-bar}_{\text{NIL}} \end{array} \right\}$
▷ 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* $\left\{ \begin{array}{l} \text{:initial-element char} \\ \text{:element-type type}_{\text{character}} \end{array} \right\}$)
▷ Return string of length *size*.

(*f*string *x*)
 $\left\{ \begin{array}{l} \text{fstring-capitalize} \\ \text{fstring-upcase} \\ \text{fstring-downcase} \end{array} \right\} x \left\{ \begin{array}{l} \text{:start start}_{\text{0}} \\ \text{:end end}_{\text{NIL}} \end{array} \right\}$
▷ Convert *x* (**symbol**, **string**, or **character**) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

$\left\{ \begin{array}{l} \text{fnstring-capitalize} \\ \text{fnstring-upcase} \\ \text{fnstring-downcase} \end{array} \right\} \widetilde{\text{string}} \left\{ \begin{array}{l} \text{:start start}_{\text{0}} \\ \text{:end end}_{\text{NIL}} \end{array} \right\}$
▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

$\left\{ \begin{array}{l} \text{fstring-trim} \\ \text{fstring-left-trim} \\ \text{fstring-right-trim} \end{array} \right\} \text{char-bag string}$
▷ Return string with all characters in sequence *char-bag* removed from both ends, from the beginning, or from the end, respectively.

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

(*f*parse-integer *string* $\left\{ \begin{array}{l} \text{:start start}_{\text{0}} \\ \text{:end end}_{\text{NIL}} \\ \text{:radix int}_{\text{10}} \\ \text{:junk-allowed bool}_{\text{NIL}} \end{array} \right\}$)
▷ 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.

$(\text{optimize} \left\{ \begin{array}{l} \text{compilation-speed}(\text{compilation-speed } n_{\text{0}}) \\ \text{debug}(\text{debug } n_{\text{0}}) \\ \text{safety}(\text{safety } n_{\text{0}}) \\ \text{space}(\text{space } n_{\text{0}}) \\ \text{speed}(\text{speed } n_{\text{0}}) \end{array} \right\})$
▷ 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.

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

(*f*encode-universal-time *sec min hour date month year* [*zone* current])
(*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.

$\left\{ \begin{array}{l} \text{flisp-implementation} \\ \text{fsoftware} \\ \text{fmachine} \end{array} \right\} \left\{ \begin{array}{l} \text{type} \\ \text{version} \end{array} \right\}$
▷ Name or version of implementation, operating system, or hardware, respectively.

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

macroexpand-hook

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

(`mtrace` {`function`
{`setf function`}^{*}})

▷ Cause *functions* to be traced. With no arguments, return list of traced functions.

(`muntrace` {`function`
{`setf function`}^{*}})

▷ Stop *functions*, or each currently traced function, from being traced.

trace-output

▷ Output stream `mtrace` and `mtime` send their output to.

(`mstep` *form*)

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

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

▷ Jump directly into debugger; return NIL. See page 38, `format`, for *control* and *args*.

(`mtime` *form*)

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

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

(`describe` *foo* [*stream* ***standard-output***])

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

(`describe-object` *foo* [*stream*])

▷ Send information about *foo* to *stream*. Called by `describe`.

(`disassemble` *function*)

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

(`room` [(NIL | default | T) default])

▷ Print information about internal storage management to ***standard-output***.

15.4 Declarations

(`proclaim` *decl*)

(`declaim` *decl*^{*})

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

(`declare` *decl*^{*})

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

(**declaration** *foo*^{*})

▷ Make *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*^{*})

(**[ftype]** *type function*^{*})

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

(**{ignorable}** {*var*
{ignore} {(**function** *function*)^{*}}})

▷ Suppress warnings about used/unused bindings.

(**inline** *function*^{*})

(**notinline** *function*^{*})

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

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

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

(`member` *foo list* {**{test function** **#'eq**
{test-not function
:key function})

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

(**{fmember-if**
fmember-if-not } *test list* [**:key function**])

▷ Return tail of *list* starting with its first element satisfying *test*. Return NIL if there is no such element.

(`subsetp` *list-a list-b* {**{test function** **#'eq**
{test-not function
:key function})

▷ Return T if *list-a* is a subset of *list-b*.

4.2 Lists

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

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

(`list*` *foo*⁺)

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

(`make-list` *num* [**:initial-element** *foo* **#'nil**])

▷ New list with *num* elements set to *foo*.

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

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

(`cdr` *list*)

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

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

(**{ffirst** | **fsecond** | **third** | **fourth** | **fifth** | **sixth** | ... | **fninth** | **tenth** } *list*)

▷ Return nth element of list if any, or NIL otherwise. **setfable**.

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

(`cXr` *list*)

▷ With *X* being one to four **as** and **ds** representing `fcars` and `fcdrs`, e.g. (`cadr bar`) is equivalent to (`car (cdr bar)`). **setfable**.

(`last` *list* [*num* **#'**]) ▷ Return list of last num conses of *list*.

(**{fbutlast** *list* } [*num* **#'**]) ▷ list excluding last *num* conses.

(**{freplace**
freplace } *cons object*)

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

(`ldiff` *list foo*)

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

(`adjoin` *foo list* {**{test function** **#'eq**
{test-not function
:key function})

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

(`pop` *place*)

▷ Set *place* to (`cdr place`), return (`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*.

{*f*subst *new* *old* *tree*}
 {*f*nsubst *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*.

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

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

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

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

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

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

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

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

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

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

15.3 REPL and Debugging

√+ | √++ | √+++
 √* | √** | √***
 √/ | √// | √///

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

√- ▷ Form currently being evaluated by the REPL.

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

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

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

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

{*f*macroexpand-1}
 {*f*macroexpand} *form* [*environment* *nil*])
 ▷ Return macro expansion, once or entirely, respectively, of *form* and *T* if *form* was a macro form. Return form and *NIL* otherwise.

cl

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

nil

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

14.4 Standard Packages

common-lisp

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

common-lisp-user

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

keyword

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

15 Compiler

15.1 Predicates

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

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

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

15.2 Compilation

(**compile** $\left\{ \begin{array}{l} \text{NIL } \text{definition} \\ \text{name} \\ \text{(setf name)} \end{array} \right\} [\text{definition}]$)

▷ 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**.

(**compile-file** *file* $\left\{ \begin{array}{l} \text{:output-file } \text{out-path} \\ \text{:verbose } \text{bool} \text{ [}*compile-verbose*] \\ \text{:print } \text{bool} \text{ [}*compile-print*] \\ \text{:external-format } \text{file-format} \text{ [default]} \end{array} \right\}$)

▷ 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**.

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

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

(**load** *path* $\left\{ \begin{array}{l} \text{:verbose } \text{bool} \text{ [}*load-verbose*] \\ \text{:print } \text{bool} \text{ [}*load-print*] \\ \text{:if-does-not-exist } \text{bool} \\ \text{:external-format } \text{file-format} \text{ [default]} \end{array} \right\}$)

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

compile-file $\left\{ \begin{array}{l} \text{pathname} \text{ [NIL]} \\ \text{truename} \text{ [NIL]} \end{array} \right\}$

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

compile $\left\{ \begin{array}{l} \text{print} \\ \text{verbose} \end{array} \right\}$

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

$\left\{ \begin{array}{l} \text{sublis } \text{association-list } \text{tree} \\ \text{fsublis } \text{association-list } \text{tree} \end{array} \right\} \left\{ \begin{array}{l} \text{:test } \text{function} \text{ [}*eq*] \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\}$

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

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

4.5 Sets

$\left\{ \begin{array}{l} \text{intersection} \\ \text{set-difference} \\ \text{union} \\ \text{set-exclusive-or} \\ \text{intersection} \\ \text{nset-difference} \\ \text{nunion} \\ \text{nset-exclusive-or} \end{array} \right\} \left\{ \begin{array}{l} a \ b \\ \tilde{a} \ b \\ \tilde{a} \ \tilde{b} \end{array} \right\} \left\{ \begin{array}{l} \text{:test } \text{function} \text{ [}*eq*] \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\}$

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

5 Arrays

5.1 Predicates

(**arrayp** *foo*)

(**vectorp** *foo*)

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

(**bit-vector-p** *foo*)

(**simple-bit-vector-p** *foo*)

(**adjustable-array-p** *array*)

(**array-has-fill-pointer-p** *array*)

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

(**array-in-bounds-p** *array* [*subscripts*])

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

5.2 Array Functions

$\left\{ \begin{array}{l} \text{make-array } \text{dimension-sizes} \text{ [:adjustable } \text{bool} \text{ [NIL]} \\ \text{adjust-array } \text{array } \text{dimension-sizes} \end{array} \right\}$

$\left\{ \begin{array}{l} \text{:element-type } \text{type} \\ \text{:fill-pointer } \{ \text{num} \text{ bool} \} \text{ [NIL]} \\ \text{:initial-element } \text{obj} \\ \text{:initial-contents } \text{tree-or-array} \\ \text{:displaced-to } \text{array} \text{ [NIL]} \text{ [:displaced-index-offset } \text{idx}] \end{array} \right\}$

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

(**aref** *array* [*subscripts*])

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

(**row-major-aref** *array* *i*)

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

(**array-row-major-index** *array* [*subscripts*])

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

(**array-dimensions** *array*)

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

(**array-dimension** *array* *i*)

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

(**array-total-size** *array*)

▷ **Number of elements** in *array*.

(**array-rank** *array*)

▷ **Number of dimensions** of *array*.

(**array-displacement** *array*)

▷ **Target array** and **offset**.

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

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

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

▷ Return result of bitwise logical operations (cf. operations of *f* **boole**, page 5) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is T, put result in *bit-array-a*; if it is NIL, make a new array for result.

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

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

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

5.3 Vector Functions

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

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

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

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

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

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

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

6 Sequences

6.1 Sequence Predicates

$$\left\{ \begin{array}{l} \text{f every} \\ \text{f notevery} \end{array} \right\} \text{ test sequence}^+$$
 ▷ Return NIL or T, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns NIL.

$$\left\{ \begin{array}{l} \text{f some} \\ \text{f notany} \end{array} \right\} \text{ test sequence}^+$$
 ▷ Return value of *test* or NIL, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns non-NIL.

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

(*f* **export** *symbols* [*package* *v**package*])
 ▷ Make *symbols* external to *package*. Return T.

(*f* **unexport** *symbols* [*package* *v**package*])
 ▷ Revert *symbols* to internal status. Return T.

$$\left\{ \begin{array}{l} \text{m do-symbols} \\ \text{m do-external-symbols} \\ \text{m do-all-symbols} \end{array} \right\} (\widehat{\text{var}} [\text{package}_{\text{v*package*}} [\text{result}_{\text{nil}}]])$$

$$(\text{declare } \widehat{\text{decl}}^*)^* \left\{ \begin{array}{l} \text{tag} \\ \text{form} \end{array} \right\}^*$$

▷ 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.

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

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

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

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

14.3 Symbols

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

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

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

(*f* **gentemp** [*prefix* *nil*] [*package* *v**package*])
 ▷ Intern fresh symbol in *package*. Deprecated.

(*f* **copy-symbol** *symbol* [*props* *nil*])
 ▷ 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*. **setf**-able.

$$\left\{ \begin{array}{l} \text{g documentation} \\ (\text{setf } \text{g documentation}) \text{ new-doc} \end{array} \right\} \text{foo} \left\{ \begin{array}{l} \text{'variable|'function} \\ \text{'compiler-macro} \\ \text{'method-combination} \\ \text{'structure|'type|'setf|T} \end{array} \right\}$$
 ▷ Get/set documentation string of *foo* of given type.

14.2 Packages

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

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

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

$(\text{mdefpackage } \text{foo} \left\{ \begin{array}{l} (:nicknames \text{nick}^*)^* \\ (:documentation \text{string}) \\ (:intern \text{interned-symbol}^*)^* \\ (:use \text{used-package}^*)^* \\ (:import-from \text{pkg} \text{imported-symbol}^*)^* \\ (:shadowing-import-from \text{pkg} \text{shd-symbol}^*)^* \\ (:shadow \text{shd-symbol}^*)^* \\ (:export \text{exported-symbol}^*)^* \\ (:size \text{int}) \end{array} \right\})$

▷ 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.

$(\text{fmake-package } \text{foo} \left\{ \begin{array}{l} (:nicknames (\text{nick}^*) \text{NIL}) \\ (:use (\text{used-package}^*)^*) \end{array} \right\})$

▷ Create *package foo*.

$(\text{frename-package } \text{package } \text{new-name} [\text{new-nicknames} \text{NIL}])$

▷ Rename *package*. Return renamed package.

$(\text{min-package } \widehat{\text{foo}})$ ▷ Make *package foo* current.

$\left\{ \begin{array}{l} \text{fuse-package} \\ \text{funuse-package} \end{array} \right\} \text{other-packages } [\text{package}_{\text{v}*\text{package}*}]$

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

$(\text{fpackage-use-list } \text{package})$

$(\text{fpackage-used-by-list } \text{package})$

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

$(\text{fdelete-package } \widehat{\text{package}})$

▷ Delete *package*. Return T if successful.

`v*package*`|`common-lisp-user` ▷ The current package.

$(\text{flist-all-packages})$ ▷ List of registered packages.

$(\text{fpackage-name } \text{package})$ ▷ Name of package.

$(\text{fpackage-nicknames } \text{package})$ ▷ Nicknames of package.

$(\text{ffind-package } \text{name})$ ▷ Package with name (case-sensitive).

$(\text{ffind-all-symbols } \text{foo})$

▷ List of symbols foo from all registered packages.

$\left\{ \begin{array}{l} \text{fintern} \\ \text{ffind-symbol} \end{array} \right\} \text{foo } [\text{package}_{\text{v}*\text{package}*}]$

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

$(\text{funintern } \text{symbol } [\text{package}_{\text{v}*\text{package}*}])$

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

$\left\{ \begin{array}{l} \text{fimport} \\ \text{fshadowing-import} \end{array} \right\} \text{symbols } [\text{package}_{\text{v}*\text{package}*}]$

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

$(\text{fshadow } \text{symbols } [\text{package}_{\text{v}*\text{package}*}])$

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

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

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

6.2 Sequence Functions

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

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

$(\text{fconcatenate } \text{type } \text{sequence}^*)$

▷ Return concatenated sequence of *type*.

$(\text{fmerge } \text{type } \widehat{\text{sequence-a}} \widehat{\text{sequence-b}} \text{test} [\text{:key } \text{function} \text{NIL}])$

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

$(\text{ffill } \widehat{\text{sequence}} \text{foo} \left\{ \begin{array}{l} (:start \text{start} \text{0}) \\ (:end \text{end} \text{NIL}) \end{array} \right\})$

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

$(\text{flength } \text{sequence})$

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

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

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

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

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

$(\text{felt } \text{sequence } \text{index})$

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

$(\text{fsubseq } \text{sequence } \text{start} [\text{end} \text{NIL}])$

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

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

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

$(\text{freverse } \text{sequence})$

$(\text{fnreverse } \widehat{\text{sequence}})$

▷ Return *sequence* in reverse order.

$\left\{ \begin{array}{l} \text{ffind} \\ \text{fposition} \end{array} \right\} \text{foo } \text{sequence} \left\{ \begin{array}{l} (:from-end \text{bool} \text{NIL}) \\ (:test \text{function} \text{#'=}) \\ (:test-not \text{test}) \\ (:start \text{start} \text{0}) \\ (:end \text{end} \text{NIL}) \\ (:key \text{function}) \end{array} \right\})$

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

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

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

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

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

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

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

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

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

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

▷ Make copy of sequence without duplicates.

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

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

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

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

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

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

$(\text{fmap } \text{type function } \text{sequence}^+)$

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

$(\text{fenough-namestring } \text{path-or-stream})$
 $[\text{root-path } \text{[*default-pathname-defaults*]}]$

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

$(\text{fnamestring } \text{path-or-stream})$
 $(\text{ffile-namestring } \text{path-or-stream})$
 $(\text{fdirectory-namestring } \text{path-or-stream})$
 $(\text{fhost-namestring } \text{path-or-stream})$

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

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

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

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

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

▷ Logical pathname of *logical-path-or-stream*. Logical pathnames are represented as all-uppercase

"[host:][;]{dir|*}+};*{name|*}*[{type|*}+]"

[.{version|*|newest|NEWEST}]"]

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

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

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

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

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

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

$(\text{fprobe-file } \text{file})$
 $(\text{ftruefilename } \text{file})$

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

$(\text{ffile-write-date } \text{file})$ ▷ Time at which *file* was last written.

$(\text{ffile-author } \text{file})$ ▷ Return name of *file* owner.

$(\text{ffile-length } \text{stream})$ ▷ Return length of *stream*.

$(\text{frename-file } \text{foo } \text{bar})$

▷ 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.

$(\text{fdelete-file } \text{file})$ ▷ Delete *file*. Return T.

$(\text{fdirectory } \text{path})$ ▷ List of pathnames matching *path*.

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

▷ 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 22.

14.1 Predicates

$(\text{fsymbolp } \text{foo})$
 $(\text{fpackagep } \text{foo})$ ▷ T if *foo* is of indicated type.
 $(\text{fkeywordp } \text{foo})$

(*m*with-output-to-string (*foo* [*string*_{NIL}[:element-type type_{character}]]))
 (declare *decl**)* *form**)
 ▷ 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.

(*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 {*host*_{NIL}:unspecific}
 :device {*device*_{NIL}:unspecific}
 :directory {({:directory}:wild_{NIL}:unspecific)
 ({:absolute}
 (:relative {directory
 :wild
 :wild-inferiors})
 :up
 :back})
 :name {*file-name*:wild_{NIL}:unspecific}
 :type {*file-type*:wild_{NIL}:unspecific}
 :version {*newest*_{version}:wild_{NIL}:unspecific}
 :defaults *path*_{host from ✓*default-pathname-defaults*}
 :case {*local*:common_{local}}}

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

{(*f*pathname-host
*f*pathname-device
*f*pathname-directory
*f*pathname-name
*f*pathname-type)
path-or-stream [:case {*local*
 :common_{local}}]

(*f*pathname-version *path-or-stream*)

▷ Return pathname component.

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

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

(*f*reduce *function* *sequence* {*:initial-value* *foo*_{NIL}
 :from-end *bool*_{NIL}
 :start *start*₀
 :end *end*_{NIL}
 :key *function*})

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

(*f*copy-seq *sequence*)

▷ Copy of sequence with shared elements.

7 Hash Tables

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

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

(*f*hash-table-p *foo*)

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

(*f*make-hash-table {*:test* {*f*eq|*f*eq|*f*equal|*f*equalp}_{≠eq}
 :size *int*
 :rehash-size *num*
 :rehash-threshold *num*})

▷ Make a hash table.

(*f*gethash *key* *hash-table* [*default*_{NIL}])

▷ Return object with *key* if any or *default* otherwise; and *T*₂ if found, *NIL*₂ otherwise. settable.

(*f*hash-table-count *hash-table*)

▷ Number of entries in *hash-table*.

(*f*remhash *key* *hash-table*)

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

(*f*clrhash *hash-table*)

▷ Empty hash-table.

(*f*maphash *function* *hash-table*)

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

(*m*with-hash-table-iterator (*foo* *hash-table*) (declare *decl**)* *form*_P*)

▷ Return values of forms. In *forms*, invocations of (*foo*) return: *T* if an entry is returned; its key; its value.

(*f*hash-table-test *hash-table*)

▷ Test function used in *hash-table*.

(*f*hash-table-size *hash-table*)

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

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

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

(*f*sxhash *foo*)

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

8 Structures

(*m*defstruct

```

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

```

▷ Define structure foo together with functions MAKE-foo, COPY-foo and foo-P; and setfable accessors foo-slot. Instances are of class foo or, if defstruct option :type is given, of the specified type. They can be created by (MAKE-foo {:slot value}*) or, if ord-λ (see page 18) is given, by (maker arg* {:key value}*). In the latter case, args and :keys correspond to the positional and keyword parameters defined in ord-λ whose vars in turn correspond to slots. :print-object/:print-function generate a gprint-object method for an instance bar of foo calling (o-printer bar stream) or (f-printer bar stream print-level), respectively. If :type without :named is given, no foo-P is created.

(*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*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)

▷ Return stream of indicated type.

(*f*make-string-input-stream string [start T] [end NIL])

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

(*f*make-string-output-stream [:element-type type character])

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

(*f*concatenated-stream-streams concatenated-stream)

(*f*broadcast-stream-streams broadcast-stream)

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

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

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

(*f*echo-stream-input-stream echo-stream)

(*f*echo-stream-output-stream echo-stream)

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

(*f*synonym-stream-symbol synonym-stream)

▷ Return symbol of synonym-stream.

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

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

(*f*file-position stream {
 :start
 :end
 position
})

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

(*f*file-string-length stream foo)

▷ Length foo would have in stream.

(*f*listen [stream v*standard-input*])

▷ T if there is a character in input stream.

(*f*clear-input [stream v*standard-input*])

▷ Clear input from stream, return NIL.

{
 *f*clear-output
 *f*force-output
 *f*finish-output
} [stream v*standard-output*])

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

(*f*close stream [:abort bool NIL])

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

(*m*with-open-file (stream path open-arg*) (declare decl*)* form^P*)

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

(*m*with-open-stream (foo stream) (declare decl*)* form^P*)

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

(*m*with-input-from-string (foo string {
 :index index
 :start start T
 :end end NIL
})) (declare

decl*)* form^P*)

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

{~ [n₀] i | ~ [n₀] :i}
 ▷ **Indent.** Set indentation to *n* relative to leftmost/to current position.

~ [c₀] [,i₀] [:] [Q] T
 ▷ **Tabulate.** Move cursor forward to column number *c* + *ki*, *k* ≥ 0 being as small as possible. With :, calculate column numbers relative to the immediately enclosing section. With Q, move to column number *c*₀ + *c* + *ki* where *c*₀ is the current position.

{~ [m₀] * | ~ [m₀] :* | ~ [n₀] Q*}
 ▷ **Go-To.** Jump *m* arguments forward, or backward, or to argument *n*.

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

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

~ [i] [:] [Q] [[{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 Q, do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.

{~?|~Q?}
 ▷ **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}*] [:] [Q] / [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.

~ [:] [Q] W
 ▷ **Write.** Print argument of any type obeying every printer control variable. With :, pretty-print. With Q, 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*open path {

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

▷ Open file-stream to *path*.

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

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

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

9.2 Variables

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

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

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

{*s*setq
*m*psq} {*symbol* *form*}*
 ▷ Set *symbols* to primary values of *forms*. Return value of last form/NIL; work sequentially/in parallel, respectively.

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

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

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

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

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

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

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

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

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

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

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

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

9.3 Functions

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

(*var** [**&optional** {(*var* [*init*_{init}] [*supplied-p*])}] [**&rest** *var*]
 [**&key** {(*var* {(*key* *var*)} [*init*_{init}] [*supplied-p*])}] [**&allow-other-keys**]
 [**&aux** {(*var* [*init*_{init}])}]^{*}]).

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

{*m*defun {*foo* (*ord-λ**)
 (*setf* *foo*) (*new-value ord-λ**) } (*declare decl**)^{*} [*doc*]
*form**)
 ▷ 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*.

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

(*s*function {*foo*
 (*m*lambda *form**)})
 ▷ Return lexically innermost function named *foo* or a lexical closure of the *mlambda* expression.

(*f*apply {*function*
 (*setf* *function*) } *arg* args*)
 ▷ Values of *function* called with *args* and the list elements of *args*. *setfable* if *function* is one of *faref*, *fbit*, and *fsbit*.

(*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.

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

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

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

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

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

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

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

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

~ [*n*_n] % ▷ **Newline**. Print *n* newlines.

~ [*n*_n] &
 ▷ **Fresh-Line**. Print *n* – 1 newlines if output stream is at the beginning of a line, or *n* newlines otherwise.

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

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

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

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

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

~ [:] [**@**] < { [*prefix*_{pre}] ~; } [*per-line-prefix* ~@:] } *body* ~; ;
 [*suffix*_{suff}] ~: [**@**] >
 ▷ **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.

✓*print-case*_(:upcase)
 ▷ Print symbol names all uppercase (:upcase), all lowercase (:downcase), capitalized (:capitalize).

✓*print-circle*_(NIL)
 ▷ If T, avoid indefinite recursion while printing circular structure.

✓*print-escape*_(NIL)
 ▷ If NIL, do not print escape characters and package prefixes.

✓*print-gensym*_(NIL) ▷ If T, print #: before uninterned symbols.

✓*print-length*_(NIL)
✓*print-level*_(NIL)
✓*print-lines*_(NIL)
 ▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

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

✓*print-pretty* ▷ If T, print prettily.

✓*print-radix*_(NIL) ▷ If T, print rationals with a radix indicator.

✓*print-readably*_(NIL)
 ▷ If T, print *readably* or signal error **print-not-readable**.

✓*print-right-margin*_(NIL)
 ▷ Right margin width in ems while pretty-printing.

(*f*set-pprint-dispatch *type function* [*priority*]
 [table_(✓*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_(✓*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_(✓*print-pprint-dispatch*)])
 ▷ Return copy of *table* or, if *table* is NIL, initial value of **✓*print-pprint-dispatch***.

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

13.5 Format

(*m*formatter *control*)
 ▷ Return *function* of *stream* and *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 **✓*standard-output***. Return NIL. If first argument is NIL, return *formatted output*.

~ [*min-col*]₍₀₎ [, [*col-inc*]₍₀₎ [, [*min-pad*]₍₀₎ [, '*pad-char*']]]
 [:] [Θ] {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 Θ, add *pad-chars* on the left rather than on the right.

~ [*radix*]₍₀₎ [, [*width*] [, '*pad-char*'] [, '*comma-char*']
 [, *comma-interval*]]] [:] [Θ] R
 ▷ Radix. (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with Θ, always prepend a sign.

(*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*.

(*f*definition {*foo*
 (setf *foo*)})
 ▷ 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*_(NIL) [*supplied-p*]]] [*E*]

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

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

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

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

or

([&whole *var*] [*E*] {*var*
 (macro-λ*)})^{*} [*E*] [&optional
 {*var*
 (macro-λ*)})^{*} [*init*_(NIL) [*supplied-p*]]] [*E*] . *rest-var*).

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

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

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

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

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

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

(**define-setf-expander** *function* (*macro*- λ^*) (**declare** \widehat{decl}^*)* [*doc*]
form^P*)
 ▷ Specify how to **setf** a place accessed by *function*. On in-
 vocation 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 **fget-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*_{INIT}])
 ▷ Return lists of temporary variables *arg-vars* and of cor-
 responding *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*.

(**define-modify-macro** *foo* ([**&optional**
 {*var*
 {(*var* [*init*_{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.

lambda-list-keywords

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

&whole *var*

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

&optional *var**

▷ Bind *vars* to corresponding arguments if any.

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

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

&key *var**

▷ Bind *vars* to corresponding keyword arguments.

&allow-other-keys

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

&environment *var*

▷ Bind *var* to the lexical compilation environment.

&aux *var**

▷ Bind *vars* as in **let**.*.

{**fwrite**
fwrite-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*_{STREAM} ***standard-output***
 }

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

(**fpprint-fill** *stream* *foo* [*parenthesis*_{INIT}] [*noop*])
 (**fpprint-tabular** *stream* *foo* [*parenthesis*_{INIT}] [*noop*] [*n*₁₀])
 (**fpprint-linear** *stream* *foo* [*parenthesis*_{INIT}] [*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
fformat directive **~/f**.

(**mpprint-logical-block** (*stream* *list* {
 {**:prefix** *string*
 :per-line-prefix *string*
 :suffix *string*_{INIT}
 }

(**declare** \widehat{decl}^*)* *form*^P*)

▷ Evaluate *forms*, which should print *list*, with *stream* lo-
 cally bound to a pretty printing stream which outputs to the
 original *stream*. If *list* is in fact not a list, it is printed by
fwrite. Return **NIL**.

(**mpprint-pop**)

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

(**fpprint-tab** {
 :line
 :line-relative
 :section
 :section-relative
 } *c* *i* [*stream*_{STREAM} ***standard-output***])

▷ Move cursor forward to column number *c* + *ki*, *k* ≥ 0
 being as small as possible.

(**fpprint-indent** {
 :block
 :current
 } *n* [*stream*_{STREAM} ***standard-output***])

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

(**mpprint-exit-if-list-exhausted**)

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

(**fpprint-newline** {
 :linear
 :fill
 :miser
 :mandatory
 } [*stream*_{STREAM} ***standard-output***])

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

print-array

▷ If T, print arrays **freadably**.

print-base₁₀

▷ Radix for printing rationals, from 2 to 36.

#+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 **v*features***, or (**{and** *or* *feature**), or (**not** *feature*).

v*features*

▷ List of symbols denoting implementation-dependent features.

|c*|; **\c**

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

13.4 Printer

**{_fprin1
_fprint
_fpprint
_fprinc}** *foo* [*stream* **v*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*pprint 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*_{NIL}
:identity *bool*{NIL}}) *form*^{P*})

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

(_fterpri [*stream* **v*standard-output***])

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

(_ffresh-line [*stream* **v*standard-output***])

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

(_fwrite-char *char* [*stream* **v*standard-output***])

▷ Output *char* to *stream*.

**{_fwrite-string
fwrite-line}** *string* [*stream* **v*standard-output***] [**{:start** *start*₀
:end *end*{NIL}}]])

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

(_fwrite-byte *byte* *stream*)

▷ Write *byte* to binary *stream*.

(_fwrite-sequence *sequence* *stream* **{_:start** *start*₀
:end *end*{NIL}})

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

9.5 Control Flow

(_sif *test* *then* [*else*_{NIL}])

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

(_mcond (*test* *then*^{P*}_{test})^{*})

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

**{_mwhen
_munless}** *test* *foo*^{P*})

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

(_mcase *test* (**{_:key** *key*^{*}) *foo*^{P*})^{*} [**{_:otherwise** *bar*^{P*}}]_{NIL}])

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

**{_mecase
mccase}** *test* (**{:key** *key*^{*}) *foo*^{P*})^{*})

▷ 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*.

(_mand *form*^P₀)

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

(_mor *form*^{*}_{NIL})

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

(_sprogn *form*^{*}_{NIL})

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

(_smultiple-value-prog1 *form-r* *form*^{*})

(_mprog1 *form-r* *form*^{*})

(_mprog2 *form-a* *form-r* *form*^{*})

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

**{_mprog
mprog*}** (**{:name** *name*
:(*name* [*value*{NIL}]))^{*} (**declare** *decl*^{*})^{*} **{_:tag** *tag*
_:*form*)^{*})

▷ Evaluate *s*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 *s*block named *NIL*.

(_sunwind-protect *protected* *cleanup*^{*})

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

(_sblock *name* *form*^{*})

▷ Evaluate *forms* in a lexical environment, and return their values unless interrupted by *s*return-from.

(_sreturn-from *foo* [*result*_{NIL}])

(_mreturn [*result*_{NIL}])

▷ Have nearest enclosing *s*block named *foo*/named *NIL*, respectively, return with values of *result*.

(_stagbody **{_:tag** *form*^{*})^{*}

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

(_sgo *tag*)

▷ Within the innermost possible enclosing *s*tagbody, jump to a tag *f*eq *tag*.

- (**scatch** *tag form*^P)
 ▷ Evaluate *forms* and return their values unless interrupted by **throw**.
- (**strow** *tag form*)
 ▷ Have the nearest dynamically enclosing **scatch** with a tag **eq tag** return with the values of *form*.
- (**sleep** *n*) ▷ Wait *n* seconds; return **NIL**.

9.6 Iteration

- (**mdo** {**mdo*** {*var* [start [step]]} }*) (stop *result*^P) (declare *decl*^{*})
 {*tag* *form*}
 ▷ Evaluate **tbody**-like body with *vars* successively bound according to the values of the corresponding *start* and *step* forms. *vars* are bound in parallel/sequentially, respectively. Stop iteration when *stop* is T. Return values of result^{*}. Implicitly, the whole form is a **block** named **NIL**.
- (**mdotimes** (var *i* [result **NTL**]) (declare *decl*^{*}) {*tag* *form*}*)
 ▷ Evaluate **tbody**-like body with *var* successively bound to integers from 0 to *i* - 1. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a **block** named **NIL**.
- (**mdolist** (var *list* [result **NTL**]) (declare *decl*^{*}) {*tag* *form*}*)
 ▷ Evaluate **tbody**-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is **NIL**. Implicitly, the whole form is a **block** named **NIL**.

9.7 Loop Facility

- (**mloop** *form*^{*})
 ▷ **Simple Loop**. If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit **block** named **NIL**.
- (**mloop** *clause*^{*})
 ▷ **Loop Facility**. For Loop Facility keywords see below and Figure 1.
- named** *n_{NTL}* ▷ Give **mloop**'s implicit **block** a name.
- with** {*var-s* (*var-s*^{*})} [*d-type*] [= *foo*]}⁺
 {**and** {*var-p* (*var-p*^{*})} [*d-type*] [= *bar*]}^{*}
 where destructuring type specifier *d-type* has the form
 {**fixnum**|**float**|**T**|**NIL**|**of-type** {*type* (*type*^{*})}}
 ▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.
- {**for**|**as**} {*var-s* (*var-s*^{*})} [*d-type*]}⁺ {**and** {*var-p* (*var-p*^{*})} [*d-type*]}^{*}
 ▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel. Destructuring type specifier *d-type* as with **with**.
- {**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*_{NTL}|*function* *#-cdr*}
 ▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

- (**fset-dispatch-macro-character** *char sub-char function* [*rt* **readtable**])
 ▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.
- (**fget-dispatch-macro-character** *char sub-char* [*rt* **readtable**])
 ▷ Dispatch function associated with *char* followed by *sub-char*.

13.3 Character Syntax

- #|** *multi-line-comment* **|#**
; *one-line-comment*
 ▷ Comments. There are stylistic conventions:
- | | |
|-----------------------|--|
| ;;; title | ▷ Short title for a block of code. |
| ;;; intro | ▷ Description before a block of code. |
| ;; state | ▷ State of program or of following code. |
| ; explanation | ▷ Regarding line on which it appears. |
| ; continuation | |
- (*foo** [*bar*_{NTL}]) ▷ List of *foos* with the terminating *cdr bar*.
- " ▷ Begin and end of a string.
- '*foo* ▷ (**quote** *foo*); *foo* unevaluated.
- `([*foo*] [*bar*] [*@baz*] [*..quux*] [*bing*])
 ▷ Backquote. **quote** *foo* and *bing*; evaluate *bar* and splice the lists *baz* and *quux* into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.
- #\c** ▷ (**character** "c"), the character *c*.
- #Bn**; **#On**; *n*.; **#Xn**; **#rRn**
 ▷ Integer of radix 2, 8, 10, 16, or *r*; $2 \leq r \leq 36$.
- n/d* ▷ The **ratio** $\frac{n}{d}$.
- { [*m*].*n* [{**S**|**F**|**D**|**L**|**E**}]*x*_{EO}] [*m*]. [*n*] [{**S**|**F**|**D**|**L**|**E**}]*x* }
 ▷ *m.n* · 10^{*x*} as **short-float**, **single-float**, **double-float**, **long-float**, or the type from ***read-default-float-format***.
- #C(a b)** ▷ (**complex** *a b*), the complex number *a + bi*.
- #'foo** ▷ (**function** *foo*); the function named *foo*.
- #nAsequence** ▷ *n*-dimensional array.
- #[n](*foo**)**
 ▷ Vector of some (or *n*) *foos* filled with last *foo* if necessary.
- #[n]*b***
 ▷ Bit vector of some (or *n*) *bs* filled with last *b* if necessary.
- #S(type {slot value}*)** ▷ Structure of *type*.
- #Pstring** ▷ A pathname.
- #:foo** ▷ Uninterned symbol *foo*.
- #.form** ▷ Read-time value of *form*.
- *,read-eval***_{NTL} ▷ 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**.

- (*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-error* *t*] [*eof-val* *nil*] [*recursive* *nil*])
 ▷ Return next character from *stream*.
- (*f* **read-char-no-hang** [*stream* *v**standard-input*] [*eof-error* *t*] [*eof-val* *nil*] [*recursive* *nil*])
 ▷ Next character from *stream* or *nil* if none is available.
- (*f* **peek-char** [*mode* *nil*] [*stream* *v**standard-input*] [*eof-error* *t*] [*eof-val* *nil*] [*recursive* *nil*])
 ▷ Next, or if *mode* is *T*, next non-whitespace character, or if *mode* is a character, next instance of it, from *stream* without removing it there.
- (*f* **unread-char** *character* [*stream* *v**standard-input*])
 ▷ Put last *f***read-char**ed *character* back into *stream*; return *nil*.
- (*f* **read-byte** [*stream* [*eof-error* *t*] [*eof-val* *nil*]])
 ▷ Read next byte from binary *stream*.
- (*f* **read-line** [*stream* *v**standard-input*] [*eof-error* *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* *: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*₁₀ ▷ 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.
- v**read-suppress*_{nil}
 ▷ If *T*, reader is syntactically more tolerant.
- (*f* **set-macro-character** *char* *function* [*non-term-p* *nil*] [*rt* *v**readtables*])
 ▷ Make *char* a macro character associated with *function* of stream and *char*. Return *T*.
- (*f* **get-macro-character** *char* [*rt* *v**readtable*])
 ▷ Reader macro function associated with *char*, and *T* if *char* is a non-terminating macro character.
- (*f* **make-dispatch-macro-character** *char* [*non-term-p* *nil*] [*rt* *v**readtables*])
 ▷ Make *char* a dispatching macro character. Return *T*.

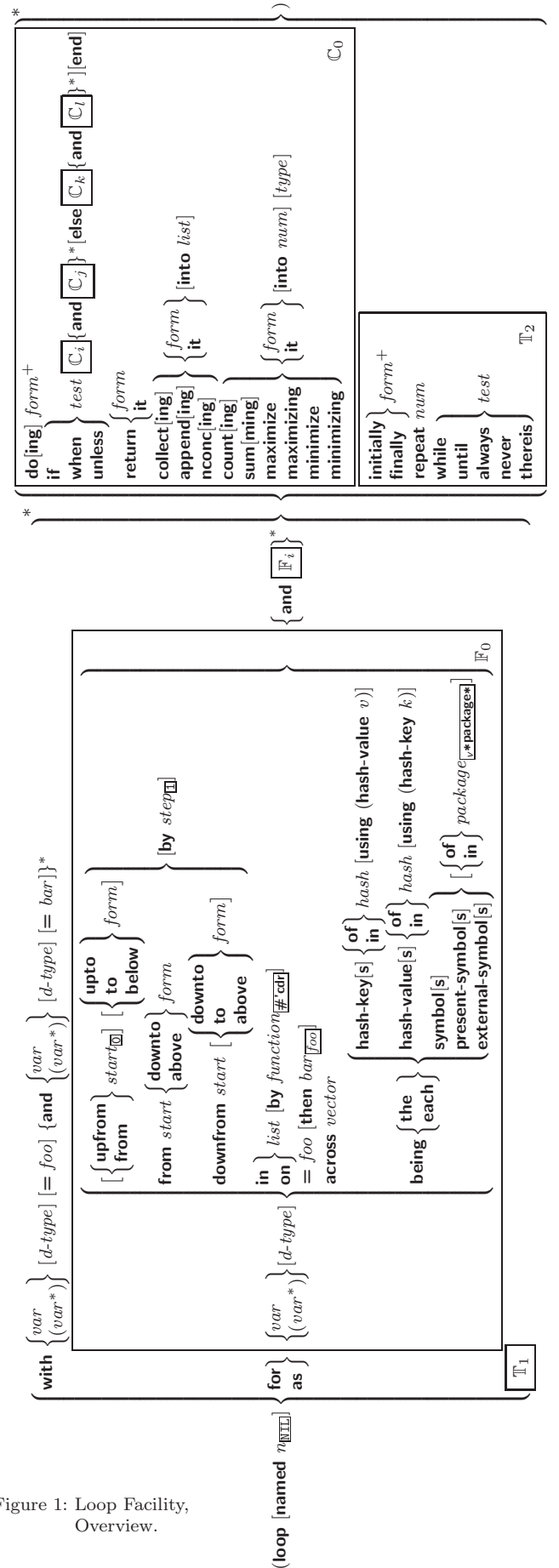


Figure 1: Loop Facility, Overview.

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

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

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

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

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

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

{do|doing} *form*⁺
 ▷ Evaluate *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 ter-
 mination.

{append|appending|nconc|nconcing} {form|it} [into *list*]
 ▷ Concatenate values of *form* or **it**, which should be lists,
 into *list* by the means of *fappend* or *fncnc*, 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 evalu-
 ated once.

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

(*mdeftype* *foo* (*macro-λ**) (*declare decl**)* [*doc*] *form*^R)
 ▷ Define type *foo* which when referenced as (*foo arg**) (or
 as *foo* if *macro-λ* doesn't contain any required parameters)
 applies expanded *forms* to *args* returning the new type. For
 (*macro-λ**) see page 19 but with default value of * instead of
 NIL. *forms* are enclosed in an implicit **block** named *foo*.

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

(*satisfies* *predicate*)
 ▷ Type specifier for all objects satisfying *predicate*.

(*mod* *n*) ▷ Type specifier for all non-negative integers < *n*.

(*not* *type*) ▷ Complement of type.

(*and* *type** □) ▷ Type specifier for intersection of *types*.

(*or* *type** □) ▷ Type specifier for union of *types*.

(*values* *type** [*&optional type** [*&rest other-args*]])
 ▷ Type specifier for multiple values.

* ▷ As a type argument (cf. Figure 2): no restriction.

13 Input/Output

13.1 Predicates

(*fstreamp* *foo*)
 (*fpathnamep* *foo*) ▷ T if *foo* is of indicated type.
 (*freadtablep* *foo*)

(*finput-stream-p* *stream*)
 (*foutput-stream-p* *stream*)
 (*finteractive-stream-p* *stream*)
 (*fopen-stream-p* *stream*)
 ▷ Return T if *stream* is for input, for output, interactive, or
 open, respectively.

(*fpathname-match-p* *path* *wildcard*)
 ▷ T if *path* matches *wildcard*.

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

13.2 Reader

{*fyes-or-no-p*
fyes-or-no-p} [*control arg**])
 ▷ Ask user a question and return T or NIL depending on their
 answer. See page 38, *fformat*, for *control* and *args*.

(*mwith-standard-io-syntax* *form*^R)
 ▷ Evaluate *forms* with standard behaviour of reader and
 printer. Return values of forms.

{*fread*
fread-preserving-whitespace} [*stream* *standard-input* [*eof-err* □
 [*eof-val* □] [*recursive* □]]])
 ▷ Read printed representation of object.

(*fread-from-string* *string* [*eof-error* □] [*eof-val* □
 [{:start *start* □
 :end *end* □
 :preserve-whitespace *bool* □}]])
 ▷ Return object read from string and zero-indexed position
 of next character.

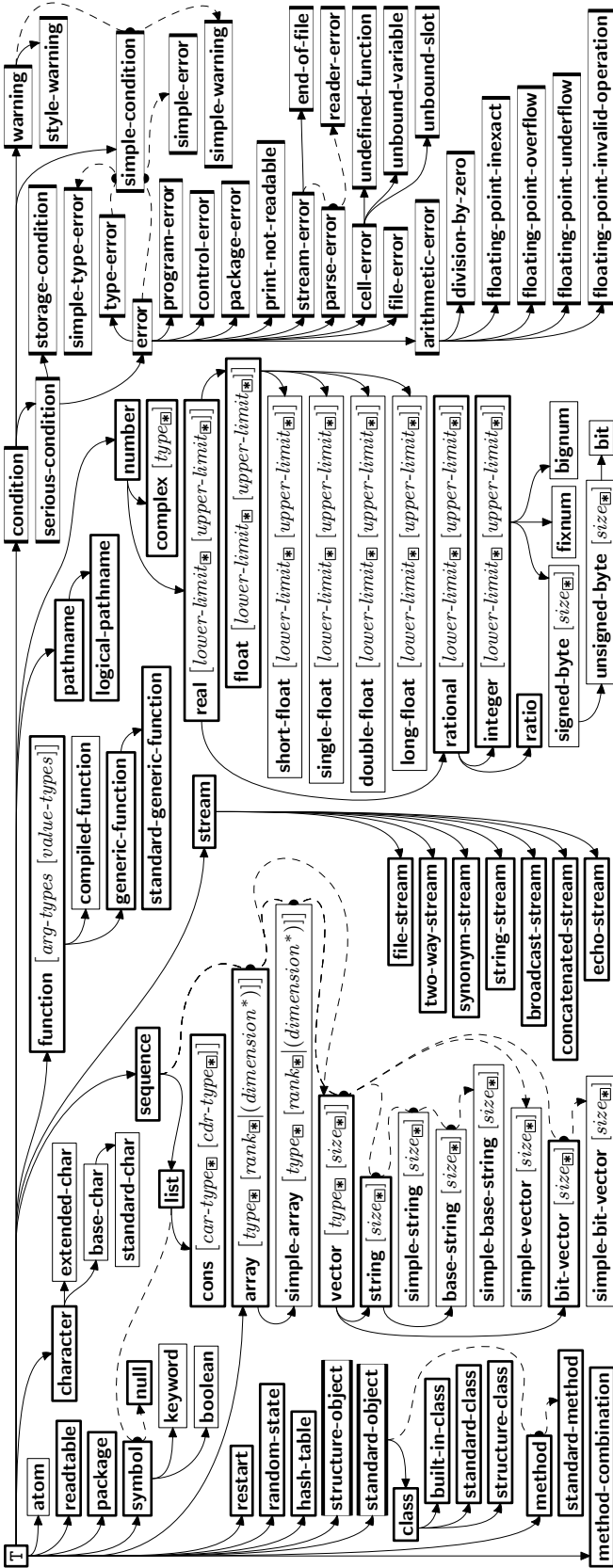


Figure 2: Precedence Order of System Classes (\square), Classes (\equiv), Types (\square), and Condition Types (\square).
Every type is also a supertype of NIL, the empty type.

{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

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

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

(*mdefclass* *foo* (*superclass** *standard-object*)

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

▷ Define or modify class *foo* as a subclass of *superclasses*. Transform existing instances, if any, by *gmake-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*.

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

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

(*greinitialize-instance* *instance* *{:initarg value}* other-keyarg**)
▷ Change local slots of *instance* according to *initargs* by means of *gshared-initialize*.

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

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

$\left\{ \begin{array}{l} \text{mwith-slots } (\widehat{\text{slot}} | (\widehat{\text{var}} \widehat{\text{slot}})^*) \\ \text{mwith-accessors } ((\widehat{\text{var}} \widehat{\text{accessor}})^*) \end{array} \right\} \text{instance (declare decl}^*)^*$
*form*_{*P*}
▷ 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**.

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

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

(**gchange-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.

(**gmake-instances-obsolete** *class*)
 ▷ Update all existing instances of *class* using **gupdate-instance-for-redefined-class**.

(**ginitialize-instance** *instance*
gupdate-instance-for-different-class *previous* *current*)
{:initarg value} other-keyarg**)
 ▷ Set slots on behalf of **gmake-instance**/of **gchange-class** by means of **gshared-initialize**.

(**gupdate-instance-for-redefined-class** *new-instance* *added-slots*
discarded-slots *discarded-slots-property-list* *{:initarg value}* other-keyarg**)
 ▷ On behalf of **gmake-instances-obsolete** and by means of **gshared-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.

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

(**gshared-initialize** *instance* *{initform-slots}* *T* *{: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.

(**gslot-missing** *class* *instance* *slot* *{setf slot-boundp slot-makunbound slot-value}* *{value}*)

(**gslot-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

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

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

(**fensure-generic-function** *{foo (setf foo)}*)

(**fcell-error-name** *condition*)
 ▷ Name of cell which caused *condition*.

(**funbound-slot-instance** *condition*)
 ▷ *Instance* with unbound slot which caused *condition*.

(**fprint-not-readable-object** *condition*)
 ▷ The *object* not readably printable under *condition*.

(**fpackage-error-package** *condition*)
(ffile-error-pathname condition)
(fstream-error-stream condition)
 ▷ *Package*, *path*, or *stream*, respectively, which caused the *condition* of indicated type.

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

(**fsimple-condition-format-control** *condition*)
(fsimple-condition-format-arguments condition)
 ▷ Return *fformat control* or list of *fformat 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.

(**ftypep** *foo* *type* *[environment NIL]*) ▷ *T* if *foo* is of *type*.

(**fsubtypep** *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.

(**sthe** *type form*) ▷ Declare *values of form* to be of *type*.

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

(**mtypecase** *foo* *(type a-form^{*})* *[({otherwise} b-form^P)]*)
 ▷ Return *values of the first a-form^{*}* whose *type* is *foo* of. Return *values of b-forms* if no *type* matches.

(**metypecase** *foo* *(type form^P)* ***)
(mctypecase *foo* *(type form^P)* ***)
 ▷ Return *values of the first form^{*}* whose *type* is *foo* of. Signal non-correctable/correctable **type-error** if no *type* matches.

(**ftype-of** *foo*) ▷ *Type* of *foo*.

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

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

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

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

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

▷ Return values of forms unless *restart* is called during their evaluation. In this case, describe *restart* using *format* *control* and *args* (see page 38) and return NIL and T.

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

(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 18 for *ord-λ**.

(*m*restart-bind ($\left\{ \begin{smallmatrix} \text{restart} \\ \text{NIL} \end{smallmatrix} \right\}$ restart-function $\left\{ \begin{smallmatrix} \text{:interactive-function arg-function} \\ \text{:report-function report-function} \\ \text{:test-function test-function} \end{smallmatrix} \right\}$ *) 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*.

(*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.

($\left\{ \begin{smallmatrix} \text{f find-restart} \\ \text{f compute-restarts name} \end{smallmatrix} \right\}$ [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.

($\left\{ \begin{smallmatrix} \text{f abort} \\ \text{f muffle-warning} \\ \text{f continue} \\ \text{f store-value value} \\ \text{f use-value value} \end{smallmatrix} \right\}$ [condition])

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

(*m*with-condition-restarts condition restarts form^P)

▷ 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*.

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

▷ 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 $\left\{ \begin{smallmatrix} \text{foo} \\ \text{(setf foo)} \end{smallmatrix} \right\}$ $\left\{ \begin{smallmatrix} \text{:before} \\ \text{:after} \\ \text{:around} \end{smallmatrix} \right\}$ $\left\{ \begin{smallmatrix} \text{primary method} \\ \text{qualifier}^* \end{smallmatrix} \right\}$)

($\left\{ \begin{smallmatrix} \text{var} \\ \text{(spec-var } \left\{ \begin{smallmatrix} \text{class} \\ \text{(eql bar)} \end{smallmatrix} \right\}) \end{smallmatrix} \right\}$ *) [*&optional*

$\left\{ \begin{smallmatrix} \text{var} \\ \text{(var [init [supplied-p]])} \end{smallmatrix} \right\}$ *) [*&rest* *var*] [*&key*

$\left\{ \begin{smallmatrix} \text{var} \\ \text{(var [init [supplied-p]])} \end{smallmatrix} \right\}$ *) [*&allow-other-keys*]

[*&aux* $\left\{ \begin{smallmatrix} \text{var} \\ \text{(var [init])} \end{smallmatrix} \right\}$ *)] $\left\{ \begin{smallmatrix} \text{(declare } \widehat{\text{decl}}^* \text{)}^* \\ \text{doc} \end{smallmatrix} \right\}$ 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.

($\left\{ \begin{smallmatrix} \text{g add-method} \\ \text{g remove-method} \end{smallmatrix} \right\}$ generic-function method)

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

(*g*find-method generic-function qualifiers specializers [error])

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

(*g*compute-applicable-methods generic-function args)

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

(*f*call-next-method arg* current args)

▷ From within a method, call next method with *args*; return its values.

(*g*no-applicable-method generic-function arg*)

▷ Called on invocation of *generic-function* on *args* if there is no applicable method. Default method signals **error**. Not to be called by user.

($\left\{ \begin{smallmatrix} \text{f invalid-method-error method} \\ \text{f method-combination-error} \end{smallmatrix} \right\}$ control arg*)

▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *args* see **format**, page 38.

(*g*no-next-method generic-function method arg*)

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

(*g*function-keywords method)

▷ Return list of keyword parameters of *method* and T if other keys are allowed.

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

10.3 Method Combination Types

standard

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

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

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

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

{
 :**documentation** *string*
 :**identity-with-one-argument** *bool*_{NIL}
 :**operator** *operator*_[c-type]
})

▷ **Short Form.** Define new **method-combination** *c-type*. In a generic function using *c-type*, evaluate most specific **:around** method supplying the values of the generic function. From within this method, **call-next-method** can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, return from the calling **call-next-method** or from the generic function, respectively, the values of (*operator* (*primary-method* *gen-arg*)*), *gen-arg** being the arguments of the generic function. The *primary-methods* are ordered [**:most-specific-first** **:most-specific-last**] (*most-specific-first*) (specified as *c-arg* in **defgeneric**). Using *c-type* as the *qualifier* in **defmethod** makes the method primary.

(**define-method-combination** *c-type* (*ord-λ**) ((*group*

{
 *
 (*qualifier** [***])
 predicate
 :**:description** *control*
 :**:order** {**:most-specific-first** **:most-specific-last**}_[most-specific-first]
 :**:required** *bool*
 :**:arguments** *method-combination-λ**
 :**:generic-function** *symbol*
 (**declare** *decl**)*
 doc
}) *body*_P*)

▷ **Long Form.** Define new **method-combination** *c-type*. A call to a generic function using *c-type* will be equivalent to a call to the forms returned by *body** with *ord-λ** bound to *c-arg** (cf. **defgeneric**), with *symbol* bound to the generic function, with *method-combination-λ** bound to the arguments of the generic function, and with *groups* bound to lists of methods. An applicable method becomes a member of the left-most *group* whose *predicate* or *qualifiers* match. Methods can be called via **call-method**. Lambda lists (*ord-λ**) and (*method-combination-λ**) according to *ord-λ* on page 18, the latter enhanced by an optional **&whole** argument.

(**call-method**

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

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

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 32.

(**define-condition** *foo* (*parent-type** **condition**)

{
 slot
 {
 :**:reader** *reader**
 :**:writer** {*writer* {**setf** *writer*}*}
 :**:accessor** *accessor**
 :**:allocation** {**:instance** {**:class** {*instance*}*}
 :**:initarg** *initarg-name**
 :**:initform** *form*
 :**:type** *type*
 :**:documentation** *slot-doc*
 }
 {
 (**default-initargs** {*name value*}*)
 (**documentation** *condition-doc*)
 (**report** {*string* *report-function*})
 }
})

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

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

▷ Return new instance of condition-type.

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

▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new instance of *condition-type* or, with **format** *control* and *args* (see page 38), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From **signal** and **warn**, return NIL.

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

▷ Unless handled, signal as correctable **error** *condition* or a new instance of *condition-type* or, with **format** *control* and *args* (see page 38), **simple-error**. In the debugger, use **format** arguments *continue-control* and *continue-args* to tag the continue option. Return NIL.

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

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

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

▷ Invoke debugger with *condition*.

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

▷ If *test*, which may depend on *places*, returns NIL, signal as correctable **error** *condition* or a new instance of *condition-type* or, with **format** *control* and *args* (see page 38), **error**. When using the debugger's continue option, *places* can be altered before re-evaluation of *test*. Return NIL.

(**handler-case** *foo* (*type* ([*var*]) (**declare** *decl**)* *condition-form*_P*)*
 [**:no-error** (*ord-λ**) (**declare** *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 18 for (*ord-λ**).

(**handler-bind** ((*condition-type* *handler-function*)* *form*_P*)

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