Contents

1 Numbers 3 9.5 Control Flow . . . . . . . 21
1.1 Predicates . . . . . . . 3 9.6 Iteration . . . . . . . 22
1.2 Numeric Functions . . . 3 9.7 Loop Facility . . . . . . . 22
1.3 Logic Functions . . . . . . 5
1.4 Integer Functions . . . 6 10 CLOS 25
1.5 Implementation- 6 Dependent . . . . . .
2 Characters 7 10.1 Classes . . . . . . . 25
3 Strings 8 10.2 Generic Functions . . . . 26
4 Conses 8 10.3 Method Combination . . .
4.1 Predicates . . . . . . . 8 Types . . . . . . . 28
4.2 Lists . . . . . . . 9
4.3 Association Lists . . . . . 10
4.4 Trees . . . . . . . 10
4.5 Sets . . . . . . . 11
5 Arrays 11
5.1 Predicates . . . . . . . 11
5.2 Array Functions . . . . . . 11
5.3 Vector Functions . . . . . 12
6 Sequences 12
6.1 Sequence Predicates . . . 12
6.2 Sequence Functions . . . . 13
7 Hash Tables 15
8 Structures 16
9 Control Structure 16
9.1 Predicates . . . . . . . 16
9.2 Variables . . . . . . . 17
9.3 Functions . . . . . . . 18
9.4 Macros . . . . . . . 19
10 CLOS 25
10.1 Classes . . . . . . . 25
10.2 Generic Functions . . . . . 26
10.3 Method Combination Types . . . . . .
11 Conditions and Errors 29
12 Types and Classes 31
13.1 Predicates . . . . . . . 33
13.2 Reader . . . . . . . 34
13.3 Character Syntax . . . . . 35
13.4 Printer . . . . . . . 36
13.5 Format . . . . . . . 38
13.6 Streams . . . . . . . 41
13.7 Pathnames and Files . . . . . 42
14 Packages and Symbols 44
14.1 Predicates . . . . . . . 44
14.2 Packages . . . . . . . 44
14.3 Symbols . . . . . . . 45
14.4 Standard Packages . . . . . 46
15 Compiler 46
15.1 Predicates . . . . . . . 46
15.2 Compilation . . . . . . . 46
15.3 REPL and Debugging . . . . . 48
15.4 Declarations . . . . . . . 49
16 External Environment 49

Typographic Conventions

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

them  
▷ Placeholder for actual code.

me  
▷ Literal text.

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

foo*; {foo}*  
▷ Zero or more foos.

foo+; {foo}+  
▷ One or more foos.

foos  
▷ English plural denotes a list argument.

{foo|bar|baz};  
▷ Either foo, or bar, or baz.

{|foo|bar|baz}  
▷ Anything from none to each of foo, bar, and baz.

ˆfoo  
▷ Argument foo is not evaluated.

˜bar  
▷ Argument bar is possibly modified.

foo*  
▷ foo* is evaluated as in :progn; see page 21.

foo: bar; baz  
▷ Primary, secondary, and nth return value.

T; NIL  
▷ t, or truth in general; and nil or ().
1 Numbers

1.1 Predicates

\( (\equiv \text{number}^+) \)  
\( (\neq \text{number}^+) \)  
\( \triangleright T \) if all numbers, or none, respectively, are equal in value.

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

\( (\minusp \ a) \)  
\( (\zerop \ a) \)  
\( (\plusp \ a) \)  
\( (\evenp \ \text{int}) \)  
\( (\oddp \ \text{int}) \)  
\( \triangleright T \) if \( a < 0 \), \( a = 0 \), or \( a > 0 \), respectively.

\( (\exp \ a) \)  
\( (\expt \ b \ p) \)  
\( (\log \ a \ [\ b \ ]) \)  
\( (\sqrt[\ a \ ]) \)  
\( (\sqrt[\ i \ ]) \)  
\( (\lcm \ \text{integer}^+) \)  
\( (\gcd \ \text{integer}^+) \)  
\( \triangleright \) Least common multiple or greatest common denominator, respectively, of integers. \( (\gcd) \) returns \( 0 \).

\( \pi \)  
\( (\sin \ a) \)  
\( (\cos \ a) \)  
\( (\tan \ a) \)  
\( (\arcsin \ a) \)  
\( (\arccos \ a) \)  
\( \triangleright \) \text{long-float} approximation of \( \pi \), Ludolph’s number.

1.2 Numeric Functions

\( (+ \ a \ b) \)  
\( (\ast \ a \ b) \)  
\( (\neg \ a \ b) \)  
\( (\div \ a \ b) \)  
\( \triangleright \) Return \( \sum a \) or \( \prod a \), respectively.

\( (+ 1 \ a) \)  
\( (\neg 1 \ a) \)  
\( \{ \text{incf} \ \text{place} \ [\delta \ ] \} \)  
\( \triangleright \) Increment or decrement the value of \( \text{place} \) by \( \delta \). Return new value.

\( (\exp \ p) \)  
\( (\expt \ b \ p) \)  
\( (\log \ a \ [\ b \ ]) \)  
\( (\sqrt[\ a \ ]) \)  
\( (\sqrt[\ i \ ]) \)  
\( (\lcm \ \text{integer}^+) \)  
\( (\gcd \ \text{integer}^+) \)  
\( \triangleright \) Least common multiple or greatest common denominator, respectively, of integers. \( (\gcd) \) returns \( 0 \).

\( \pi \)  
\( (\sin \ a) \)  
\( (\cos \ a) \)  
\( (\tan \ a) \)  
\( (\arcsin \ a) \)  
\( (\arccos \ a) \)  
\( \triangleright \) \text{arcsin} \( a \), \text{arccos} \( a \), respectively. \( (\ a \ \text{in radians}) \).
\( r \text{atan } a \) \[ \frac{\pi}{4} \text{ in radians.} \]

\( r \text{sinh } a \)
\( r \text{cosh } a \)
\( r \text{tanh } a \)
\( r \text{asinh } a \)
\( r \text{acosh } a \)
\( r \text{atanh } a \)
\( r \text{cis } a \)
\( r \text{conjugate } a \)
\( r \text{max } a \)
\( r \text{min } a \)
\[
\begin{align*}
\{\text{round} \} & n \in \mathbb{Q} \\
\{\text{floor} \} & n \in \mathbb{R} \\
\{\text{ceiling} \} & n \in \mathbb{Q} \\
\{\text{truncate} \} & n \in \mathbb{R} \\
\end{align*}
\]
\( r \text{mod } a \)
\( r \text{rem } a \)
\( r \text{random } a \)
\( r \text{make-random-state } a \)

\( r \text{random-state} \)
\( r \text{float-sign } a \)
\( r \text{signum } a \)
\( r \text{numerator } a \)
\( r \text{denominator } a \)
\( r \text{realpart } a \)
\( r \text{imagpart } a \)
\( r \text{complex } a \)
\( r \text{phase } a \)
\( r \text{abs } a \)
\( r \text{rational } a \)
\( r \text{rationalize } a \)
\( r \text{float } a \)

Common Lisp Quick Reference
Negative integers are used in two's complement representation.

(boole operation int-a int-b)  
\( \downarrow \) Return value of bitwise logical operation. operations are

- boole-1  \( \downarrow \) int-a.
- boole-2  \( \downarrow \) int-b.
- boole-c1  \( \downarrow \) int-a \( \equiv \) int-b.
- boole-c2  \( \downarrow \) int-b.
- boole-set  \( \downarrow \) All bits set.
- boole-clr  \( \downarrow \) All bits zero.
- boole-eqv  \( \downarrow \) int-a \( \equiv \) int-b.
- boole-and  \( \downarrow \) int-a \& int-b.
- boole-andc1  \( \downarrow \) int-a \& int-b.
- boole-andc2  \( \downarrow \) \neg (int-a \& int-b).
- boole-or  \( \downarrow \) int-a \lor int-b.
- boole-orc1  \( \downarrow \) int-a \lor int-b.
- boole-orc2  \( \downarrow \) \neg (int-a \lor int-b).
- boole-xor  \( \downarrow \) \neg (int-a \& \neg int-b).

\((-\text{lognot})\) integer  \( \downarrow \) \neg integer.

\((-\text{logeqv})\) integer*  \( \downarrow \) Return value of exclusive-nored or added integers, respectively. Without any integer, return \(-1\).

\((-\text{logand})\) integer*  \( \downarrow \) \neg (int-a \& int-b).

\((-\text{logandc1})\) int-a int-b  \( \downarrow \) \neg (int-a \& \neg int-b).

\((-\text{logandc2})\) int-a int-b  \( \downarrow \) \neg (int-a \& \neg int-b).

\((-\text{logand})\) int-a int-b  \( \downarrow \) \neg (int-a \& \neg int-b).

\((-\text{logxor})\) integer*  \( \downarrow \) Return value of exclusive-orred or orred integers, respectively. Without any integer, return \(0\).

\((-\text{logor})\) integer*  \( \downarrow \) \neg (int-a \lor int-b).

\((-\text{logorc1})\) int-a int-b  \( \downarrow \) \neg (int-a \lor int-b).

\((-\text{logorc2})\) int-a int-b  \( \downarrow \) \neg (int-a \lor int-b).

\((-\text{logior})\) integer*  \( \downarrow \) Return value of exclusive-ored or orred integers, respectively. Without any integer, return \(0\).

\(-\text{logb} itp\) int  \( \downarrow \) If zero-indexed ith bit of int is set.

\(-\text{logtest}\) int-a int-b  \( \downarrow \) Return \(T\) if there is any bit set in int-a which is set in int-b as well.

\(-\text{logcount}\) int  \( \downarrow \) Number of 1 bits in int \(\geq 0\), number of 0 bits in int < 0.
1.4 Integer Functions

\[(\text{integer-length} \ n)\]  
\(\rightarrow\) Number of bits necessary to represent \(n\).

\[(\text{ldb-test} \ \text{byte-spec} \ n)\]  
\(\rightarrow\) Return \(1\) if any bit specified by \(\text{byte-spec}\) in \(n\) is set.

\[(\text{ash} \ n \ \text{count})\]  
\(\rightarrow\) Return copy of \(n\) arithmetically shifted left by \(\text{count}\) adding zeros at the right, or, for \(\text{count} < 0\), shifted right discarding bits.

\[(\text{byte-size} \ n)\]  
\(\rightarrow\) Extract byte denoted by \(\text{byte-spec}\) from \(n\). \text{setf}\table.

\[(\text{deposit-field} \ \text{db})\]  
\(\rightarrow\) int-b with bits denoted by \(\text{byte-spec}\) replaced by corresponding bits of int-a, or by the low \(\text{byte-size} \ \text{byte-spec}\) bits of int-a, respectively.

\[(\text{mask-field} \ \text{byte-spec} \ int)\]  
\(\rightarrow\) Return copy of int with all bits unset but those denoted by \(\text{byte-spec}\). \text{setf}\table.

\[(\text{byte-size} \ n)\]  
\(\rightarrow\) Byte specifier for a byte of \(n\) bits starting at a weight of 0 or, +0, respectively.

\[(\text{byte-size} \ n)\]  
\(\rightarrow\) Size or position, respectively, of \(\text{byte-spec}\).

1.5 Implementation-Dependent

\[(\text{short-float} \ \text{single-float} \ \text{epsilon} \ \text{negative-epsilon} \ \text{long-float})\]  
\(\rightarrow\) Smallest possible number making a difference when added or subtracted, respectively.

\[(\text{least-negative} \ \text{least-negative-normalized} \ \text{least-positive} \ \text{least-positive-normalized} \ \text{long-float})\]  
\(\rightarrow\) Available numbers closest to \(-0\) or \(+0\), respectively.

\[(\text{most-negative} \ \text{most-positive} \ \text{double-float} \ \text{long-float} \ \text{fixnum})\]  
\(\rightarrow\) Available numbers closest to \(-\infty\) or \(+\infty\), respectively.

\[(\text{decode-float} \ n)\]  
\(\rightarrow\) Significand, exponent, and sign of \(n\). 

\[(\text{scale-float} \ n)\]  
\(\rightarrow\) With \(n\)’s radix \(b\), return \(n^b\).

\[(\text{float-radix} \ n)\]  
\(\rightarrow\) Radix, number of digits in that radix, or precision in that radix, respectively, of \(n\).

\[(\text{upgraded-complex-part-type} \ \text{foo} \ \text{environment} \ \text{env})\]  
\(\rightarrow\) Type of most specialized \text{complex} number able to hold parts of \(\text{type\ foo}\).
2 Characters

The standard-character type comprises a-z, A-Z, 0-9, Reovline, Space, and !?*+,:;\_-.<>@#%^&() [[]].

(characterp foo) ➔ T if argument is of indicated type.

(graphic-char-p character) ➔ T if character is visible, alphabetic, or alphanumeric, respectively.

(upper-case-p character) ➔ T if character is uppercase, lowercase, or able to be in another case, respectively.

(lower-case-p character) ➔ T if character is equal to its lowercase, respectively.

(both-case-p character) ➔ T if character is monotonically decreasing, monotonically non-increasing, or monotonically non-decreasing, respectively.

(standard-char-p character) ➔ T if character is a digit, or NIL otherwise.

(otherwise).
3 Strings

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

(stringp foo) → ⊤ if foo is of indicated type.

(simple-string-p foo) → ⊤ if foo is of indicated type.

{string = foo bar} :size
{string-equal foo bar} → ⊤ if subsequences of foo and bar are equal. Obey/ignore, respectively, case.

{string /= not-equal} :string-size
{string >= greaterp} :string-size
{string > not-lessp} :string-size
{string < lessp} :string-size
{string <= not-greaterp} :string-size
→ 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.

(make-string :size :initial-element char
:element-type type :character) → Return string of length size.

{string x} :string-size
{string-upcase x} :string-size
{string-downcase x} :string-size
→ Convert x (symbol, string, or character) into a string, a string with capitalized words, an all-upper case string, or an all-lower case string, respectively.

{string-capitalize string} :string-size
{string-upcase string} :string-size
{string-downcase string} :string-size
→ Convert string into a string with capitalized words, an all-upper case string, or an all-lower case string, respectively.

{string-trim char-bag string} :string-size
{string-left-trim char-bag string} :string-size
{string-right-trim char-bag string} :string-size
→ Return string with all characters in sequence char-bag removed from both ends, from the beginning, or from the end, respectively.

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

(parse-integer string radix integer) :string-size
→ Return integer parsed from string and index of parse end.

4 Conses

4.1 Predicates

(consp foo) → ⊤ if foo is of indicated type.

(listp foo) → ⊤ if foo is of indicated type.

(endp list) :null
(null foo) :null
→ Return ⊤ if list/foo is NIL.

15.4 Declarations

(proclaim 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, ignore, ignore, ignore, inline, notinline, optimize, or special. See below.

(directive) → 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 variable*) → Declare variables or functions to be of type.

(declare variable* (function function*)) → Suppress warnings about used/unused bindings.

(line function*) → Tell compiler to integrate/not to integrate, respectively, called functions into the calling routine.

(compile-time function*) → 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

(get-real-time) → Current time, or computing time, respectively, in clock ticks.

(get-time-units-per-second) → Number of clock ticks per second.

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

(get-universal-time) → Seconds from 1900-01-01, 00:00, ignoring leap seconds.

(decode-universal-time universal-time [time-zone]) → Return second, minute, hour, date, month, year, day, daylight-p, and zone.

(short-site-name) → String representing physical location of computer.

(long-site-name) → String representing physical location of computer.

(lisp-implementation software machine type* version*) → Name or version of implementation, operating system, or hardware, respectively.

(machine-instance) → Computer name.
**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.

- `v` (/ /) (/)
  - Form currently being evaluated by the REPL.

- `(apropos string [package])`
  - Print interned symbols containing string.

- `(apropos-list string [package])`
  - List of interned symbols containing string.

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

- `(red [file-or-function])`  > Invoke editor if possible.

- `{macroexpand-1}`
  - Macroexpand: Return macro expansion, once or entirely, respectively, of form and T if form was a macro form. Return form and NIL otherwise.

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

- `(mtrace function)`
  - Cause functions to be traced. With no arguments, return list of traced functions.

- `(muntrace function)`
  - Stop tracing, or each currently traced function, from being traced.

- `(trace-output)`
  - Output stream `mtrace` and `mtime` send their output to.

- `(step form)`
  - Step through evaluation of form. Return values of form.

- `(break [control arg]*)`
  - Jump directly into debugger; return NIL. See page 38, for format, for control and args.

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

- `(inspect foo)`
  - Interactively give information about foo.

- `(describe foo)`
  - Send information about foo to stream.

- `(describe-object foo)`
  - 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] [object])`
  - Print information about internal storage management to `standard-output`.

---

**4.2 Lists**

- `(atom foo)`  > Return T if foo is not a cons.

- `(tailp list)`  > Return T if foo is a tail of list.

- `(member foo list)`
  -成员函数function
  -成员函数not function
  -key function
  -Return tail of list starting with its first element matching foo. Return NIL if there is no such element.

- `(member-if)`
  -成员函数if
  -member-if-not function
  -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)`
  -subset function
  -test function
  -member-if-not function
  -member-if-not function
  -key function
  -Return T if list-a is a subset of list-b.

- `(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])`
  -Return 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. settable.

- `(cdr list)`  > Cdr of list or NIL if list is NIL. settable.

- `(rest list)`  > Return n-th element of list if any, or NIL otherwise. settable.

- `(nth n list)`  > Zero-indexed n-th element of list. settable.

- `(cadr list)`  > cadr of list.

- `(cadr list)`  > With X being one to four as and ds representing cads and cdrs, e.g. (cadr bar) is equivalent to (car (cdr bar)).
  -settable.

- `(last list [num])`  > Return list of last num conses of list.

- `(butlast list)`
  -Return list excluding last num conses.

- `(rplaca)`
  -Replace car, or cdr, respectively, of cons with object.

- `(rplcad)`
  -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)`
  -adjoin function
  -test 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).
Common Lisp Quick Reference

4.4 Trees

(tree-equal foo bar) ➤ Return T if trees foo and bar have same shape and leaves satisfying test.

{tree-equal foo bar} //test function
{tree-equal} //test-not function
{key function}

(subst new old tree) ➤ Make copy of tree with each subtree or leaf matching old replaced by new.

{subst new old tree} //test function
{subst new old tree} //test-not function
{key function}

(subst-if-not new test tree) ➤ Make copy of tree with each subtree or leaf satisfying test replaced by new.

{subst-if-not new test tree} //test function
{subst-if-not new test tree} //test-not function
{key function}

4.3 Association Lists

(pairlis keys values alist) ➤ Prepend to alist an association list made from lists keys and values.

{pairlis keys values} //alist
{pairlis} //key

(acons key value alist) ➤ Return alist with (key, value) pair added.

{acons key} //value
{acons} //alist

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

{mapcar} //function
{mapcar list+}

(maplist 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.

{maplist} //function
{maplist list+}

(mapcan 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.

{mapcan} //function
{mapcan list+}

(copy-list list) ➤ Return copy of list with shared elements.
46

5 Arrays

5.1 Predicates

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

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

5.2 Array Functions

(make-array dimension-sizes [adjustable bool])  ▷ Returns a fresh, uninterned symbol.

(adj-adjust array dimension-sizes)

| element-type type |
| fill-pointer { num bool } |
| :initial-object |
| :initial-contents tree-or-array |
| :displaced-to array | :displaced-index-offset |

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

(aref array [subscripts])  ▷ Return array element pointed to by subscripts, if felsible.

(row-major-aref array i)  ▷ Return i-th element of array in row-major order, if felsible.

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

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

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

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

(array-rank array)  ▷ Number of dimensions of array.
(array-displacement array)  > Target array and offset.

(bit bit-array [subscripts])

(sbit simple-bit-array [subscripts])

(bit-not bit-array [result-bit-array])

(bit-eqv)

(bit-and)

(bit-anc1)

(bit-anc2)

(bit-nand)

(bit-ior)

(bit-orc1)

(bit-orc2)

(bit-xor)

(bit-nor)

> Return element of bit-array or of simple-bit-array. setfable.

> 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.

> Return result of bitwise logical operations (cf. opera-
tions of boolean, 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.

> Upper bound of array rank; ≥ 8.

> Upper bound of an array dimension; ≥ 1024.

> Upper bound of array size; ≥ 1024.

5.3 Vector Functions

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

(vector foo)

(svref vector i)

(vector-push foo vector)

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

(vector-pop vector)

(fill-pointer vector)

6 Sequences

6.1 Sequence Predicates

(every test sequence)

(not every)

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

14.3 Symbols

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

(make-symbol name)

> Make fresh, uninterned symbol name.
14 Packages and Symbols

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

14.1 Predicates

(symbolp foo)  →  T if foo is of indicated type.

(packagep foo)  →  T if foo is a package.

(keywordp foo)  →  T if foo is a keyword.

14.2 Packages


package:symbol  →  Exported symbol of package.

package:symbol  →  Possibly unexported symbol of package.

(make-package foo)  →  Create or modify package foo.


(mk-package foo)  →  Make package foo current.

(use-package)  →  Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T if successful.

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

(delete-package package)  →  Delete package. Return T if successful.

*package*  →  The current package.

(list-all-packages)  →  List of registered packages.

(package-name package)  →  Name of package.

(package-nicknames package)  →  Nicknames of package.

(reverse sequence)  →  Return sequence in reverse order.

Common Lisp Quick Reference

14.1 Predicates

(symbolp foo)  →  T if foo is of indicated type.

(packagep foo)  →  T if foo is a package.

(keywordp foo)  →  T if foo is a keyword.

14.2 Packages


package:symbol  →  Exported symbol of package.

package:symbol  →  Possibly unexported symbol of package.

(make-package foo)  →  Create or modify package foo.


(mk-package foo)  →  Make package foo current.

(use-package)  →  Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T if successful.

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

(delete-package package)  →  Delete package. Return T if successful.

*package*  →  The current package.

(list-all-packages)  →  List of registered packages.

(package-name package)  →  Name of package.

(package-nicknames package)  →  Nicknames of package.

(reverse sequence)  →  Return sequence in reverse order.
\(\text{Common Lisp Quick Reference}\)

\[
\begin{aligned}
\{\text{find} \atop \text{position}\} & \quad \text{foo sequence} \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{find-if} \atop \text{if-not}\} & \quad \text{test sequence} \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{search} \atop \text{sequence-a sequence-b}\} & \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{remove} \atop \text{foo sequence}\} & \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

- Make copy of sequence without elements matching foo.

\[
\begin{aligned}
\{\text{remove-if} \atop \text{if-not}\} & \quad \text{test sequence} \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{remove-duplicates} \atop \text{sequence}\} & \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

- Make copy of sequence without duplicates.

\[
\begin{aligned}
\{\text{substitute} \atop \text{new old sequence}\} & \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

- Make copy of sequence with all (or count) old elements replaced by new.

\[
\begin{aligned}
\{\text{substitute-if} \atop \text{if-not}\} & \quad \text{new test sequence} \\
\{\text{test function}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{merge-pathnames} \atop \text{path-or-stream}\} & \\
\{\text{default-pathname-defaults}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{user-home-dir} \atop \text{pathname}\} & \\
\{\text{host}\} & \quad \text{function}
\end{aligned}
\]

- User’s home directory.

\[
\begin{aligned}
\{\text{enough-namestring} \atop \text{path-or-stream}\} & \\
\{\text{root-path}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{namestring} \atop \text{path-or-stream}\} & \\
\{\text{file-namestring}\} & \quad \text{function}
\end{aligned}
\]

- Translate the path of path-or-stream into wildcard-path-a.

\[
\begin{aligned}
\{\text{host-namestring} \atop \text{path-or-stream}\} & \\
\{\text{host}\} & \quad \text{function}
\end{aligned}
\]

- Pathname of path-or-stream.

\[
\begin{aligned}
\{\text{translate-pathname} \atop \text{path-or-stream}\} & \\
\{\text{wildcard-path-a}\} & \quad \text{function}
\end{aligned}
\]

- Logical pathname of logical-path-or-stream. Logical pathnames are represented as all-uppercase \("[\text{host}:][|\{\text{dir}\}|^*];|\{\text{name}\}|^*][|\{\text{type}\}|^*]\{\text{version}\}^*\). If host is unspecified, host is "LISP".

\[
\begin{aligned}
\{\text{load-logical-pathnames}\} & \\
\{\text{logical-host}\} & \quad \text{function}
\end{aligned}
\]

- Logical pathnames for logical-host. Return empty list if already loaded; return T if successful.

\[
\begin{aligned}
\{\text{translate-logical-pathname} \atop \text{path-or-stream}\} & \\
\{\text{logical-path-or-stream}\} & \quad \text{function}
\end{aligned}
\]

- Pathname of path-or-stream.

\[
\begin{aligned}
\{\text{probe-file}\} & \\
\{\text{file}\} & \quad \text{function}
\end{aligned}
\]

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

\[
\begin{aligned}
\{\text{file-write-date}\} & \\
\{\text{file}\} & \quad \text{function}
\end{aligned}
\]

- Time at which file was last written.

\[
\begin{aligned}
\{\text{file-author}\} & \\
\{\text{file}\} & \quad \text{function}
\end{aligned}
\]

- Return name of file owner.

\[
\begin{aligned}
\{\text{file-length}\} & \\
\{\text{stream}\} & \quad \text{function}
\end{aligned}
\]

- Length of stream.

\[
\begin{aligned}
\{\text{rename-file}\} & \\
\{\text{foo bar}\} & \quad \text{function}
\end{aligned}
\]

- 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.

\[
\begin{aligned}
\{\text{delete-file}\} & \\
\{\text{file}\} & \quad \text{function}
\end{aligned}
\]

- Delete file. Return T.
13.7 Pathnames and Files

- **host**
  - NIL: unspecific
- **device**
  - NIL: unspecific
- **directory**
  - NIL: unspecific
- **:name**
  - NIL: unspecific
- **:type**
  - NIL: unspecific
- **:version**
  - NIL: unspecific
- **:defaults**
  - path: host from default-pathname-directories

**Construct a logical pathname from a physical pathname.** For `:case` `:local`, leave case of components unchanged. For `:case` `:common`, leave mixed-case components unchanged; convert all-upper case components into local customary case; do the opposite with all-lowercase components.

17 of external entries in hash-table of NIL and T form:

- :wild
- :wild-inferiors
- :up
- :back

- **string-stream**
  - forms with foo locally bound to string. Return values of forms.

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

- **stream-external-format**
  - 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.

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.

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

- **/make-hash-table**

- **/gethash**
  - Return object with key if any or `default` otherwise; and T if found, NIL otherwise. `set-table`.

- **/hash-table-count**
  - Number of entries in hash-table.

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

- **/chash**
  - Empty hash-table.

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

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

- **/hash-table-test**
  - Test function used in hash-table.

- **/hash-table-size**
- **/hash-table-rehash-size**
- **/hash-table-rehash-threshold**
8 Structures

(defstruct foo
    [:concrete-slot
     slot [slot-type (read-only)]][
        :concrete-name
        [slot-prefix]]
    :copier
    copier
    :copy
    [slot
     [slot-init
      :type sl-type
      :read-only b]]
    :doc
    [slot
     slot-init
     :type sl-type
     :read-only b]
)  
> Define structure foo together with functions MAKE-foo, COPY-foo and foo-P, and setfable accessor 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 ears in turn correspond to slots. :print-object / :print-function generate a :print-object method for an instance bar of foo calling (o-printer bar stream) or (f-printer bar stream print-level), respectively. If :type without :named is given, no foo-P is created.

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

9 Control Structure

9.1 Predicates

(eq foo bar)  
> T if foo and bar are identical.

(eql foo bar)  
> T if foo and bar are identical, or the same character, or numbers of the same type and value.

(equal foo bar)  
> T if foo and bar are eql, or are equivalent pathnames, or are conses with eql cars and cdrs, or are strings or bit-vectors with eql elements below their fill pointers.

(equalp foo bar)  
> T if foo and bar are identical; or are the same character ignoring case; or are numbers of the same value ignoring type; or are equivalent pathnames; or are conses or arrays of the same shape with equalp elements; or are structures of the same type with equalp elements; or are hash-tables of the same size with the same :test function, the same keys in terms of :test function, and equalp elements.
- \[[\text{1}][\text{0}] < \{(prefix \text{-} \text{-})\} \text{ per-line-prefix -\text{0}}\}\] body \text{-}; suffix \text{-}; \text{-}; \text{-}; \text{-};
  \[\text{Logical Block.}\] Act like \text{print-logical-block} using body as \text{format} control string on the elements of the list argument or, with \text{1}, the remaining arguments, which are extracted by \text{print-pop}. With \text{-}, \text{prefix} and \text{suffix} default to \text{ (} and \text{)}. When closed by \text{-}; \text{-};, spaces in \text{body} are replaced with conditional newlines.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{d}\}
\]
  \[\text{Indent.}\] Set indentation to \text{n} relative to leftmost/to current position.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{t}\}
\]
  \[\text{Tabulate.}\] Move cursor forward to column number \(c + k_i\), \(k \geq 0\) as being small as possible. With \text{:}, calculate column numbers relative to the immediately enclosing section. With \text{0}, move to column number \(c_0 + c + k_i\) where \(c_0\) is the current position.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{m}\}
\]
  \[\text{Go-To.}\] Jump \(m\) arguments forward, or backward, or to argument \(n\).

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{f}\}
\]
  \[\text{Iteration.}\] Use \text{text} repeatedly, up to \text{limit}, as control string for the elements of the list argument or with \text{0} for the remaining arguments. With \text{:}, \text{list} elements or remaining arguments should be lists of which a new one is used at each iteration step.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{x}\}
\]
  \[\text{Escape Upward.}\] Leave immediately \text{-}; \text{-}; \text{-}; \text{-}; \text{-}; \text{-};, or the entire \text{format} operation. With \text{e} for \text{three} prefixes, act only if \(x = 0\), \(x = y\), or \(x \leq y \leq z\), respectively.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{b}\}
\]
  \[\text{Conditional Expression.}\] Use the zero-indexed argument (or \text{ith} if given) \text{text} as a \text{format} control subclause. With \text{:}, use the first \text{text} if the argument value is \text{NIL}, or the second \text{text} if it is \text{T}. With \text{0}, do nothing for an argument value of \text{NIL}. Use the only \text{text} and leave the argument to be read again if it is \text{T}.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{0}\}
\]
  \[\text{Recursive Processing.}\] 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.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{f}\}
\]
  \[\text{Call Function.}\] Call all-uppercase \text{function} with the arguments stream, \text{format-argument}, \text{colon-p}, \text{at-sign-p} and \text{prefixes} for printing \text{format-argument}.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{w}\}
\]
  \[\text{Write.}\] Print argument of any type obeying every printer control variable. With \text{0}, \text{pretty-print}. With \text{1}, print without limits on length or depth.

\[
\{- \mid \text{[\text{1}]} \mid \text{[\text{0}]} \mid \text{i} \mid \text{#}\}
\]
  \[\text{In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.}\]
9.3 Functions

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

\[
(var \text{ \&optional } var \text{ \&rest } var)
\]

\[
(var \text{ \&key } var \text{ \&allow-other-keys})
\]

\[
(var \text{ \&aux } var)
\]

\[
supplied-p = T \text{ if there is a corresponding argument. } init \text{ forms can refer to any } init \text{ and supplied-p to their left.}
\]

\[
defun \text{ forms are enclosed in an implicit block named } foo.
\]

\[
(let \text{ forms with locally defined functions } foo. \text{ Defined functions of the same name are shadowed. Each } foo \text{ is also the name of an implicit block around its corresponding local-form*.}
\]

\[
(let \text{ functions } foo \text{ are visible inside local-forms.}
\]

\[
function \text{ forms are enclosed in the } \lambda \text{ expression.}
\]

\[
(function \text{ elements are } \lambda \text{.
\]

\[
(apply \text{ values of } function \text{ called with } args \text{ and the list elements of } args. \text{ Settable if function is one of } \makebox{setf}, \makebox{setit}, \text{ and } \makebox{sbset}.
\]

\[
(funcall \text{ values of } function \text{ called with } args.
\]

\[
(multiple-value-call \text{ call with all the values of each form as its arguments.}
\]

\[
(values-list \text{ return values by function.}
\]

\[
(values \text{ return as multiple values the primary values of the } foo.
\]

\[
(multiple-value-list \text{ list of the values of form.}
\]

- \[\text{radix} \text{ \&width} \text{ \&pad-char} \text{ \&comma-char} \text{ \&comma-integer} \text{ \&comma-fraction} \text{ \&comma-mixed} \text{ \&comma-scientific} \text{ \&comma-compact} \]

- \[\text{Radix.} \text{ Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.}
\]

- \[\text{width} \text{ \&pad-char} \text{ \&comma-char} \text{ \&comma-integer} \text{ \&comma-fraction} \text{ \&comma-mixed} \text{ \&comma-scientific} \text{ \&comma-compact} \]

- \[\text{Fixed-Format Floating-Point.} \text{ With } \\& \text{ always prepend a sign.}
\]

- \[\text{Monetary Floating-Point.} \text{ Print argument as fixed-format floating-point number. With } \\& \text{ put sign before any padding; with } \\& \text{ always prepend a sign.}
\]

- \[\text{Character.} \text{ Print, spell out, print in } \text{#\标明} \text{ syntax, or tell how to type, respectively, arguments as (possibly non-printing) character.}
\]

- \[\text{Case-Conversion.} \text{ Convert text to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.}
\]

- \[\text{Plural.} \text{ If argument } eq\标明 1 \text{ print nothing, otherwise print } s; \text{ do the same for the previous argument; if argument } eq\标明 1 \text{ print } y, \text{ otherwise print } ies; \text{ do the same for the previous argument, respectively.}
\]

- \[\:\% \text{ Newline.} \text{ Print } n \text{ newlines.}
\]

- \[\:\& \text{ Fresh-Line.} \text{ Print } n \text{ newlines if output stream is at the beginning of a line, or } n \text{ newlines otherwise.}
\]

- \[\:\| \text{ Conditional Newline.} \text{ Print a newline like } \text{pprint-newline} \text{ with argument } \text{linear}, \text{ fill}, \text{ minis}, \text{ and } \text{mandatory}, \text{ respectively.}
\]

- \[\:\text{t} \text{ Ignored Newline.} \text{ Ignore newline, or whitespace following newline, or both, respectively.}
\]

- \[\:\text{Page.} \text{ Print } n \text{ page separators.}
\]

- \[\:\text{Title.} \text{ Print } n \text{ titles.}
\]

- \[\text{Justification.} \text{ Justify text produced by } \text{texts} \text{ in a field of at least } min-col \text{ columns. With } \:\text{, right justify; with } \\& \text{, left justify. If this would leave less than } \text{spare characters on the current line, output } \text{nl-text first.}
\*print-array\*  \> If \(T\), print arrays \(\text{readably}\).

\*print-base\* \([\text{base}]\)  \> Radix for printing rationals, from 2 to 36.

\*print-case\* \([\text{case}]\)  \> Print symbol names all uppercase \(\text{(upcase)}\), all lowercase \(\text{(downcase)}\), capitalized \(\text{(capitalize)}\).

\*print-circle\* \([\text{NIL}]\)  \> If \(T\), avoid indefinite recursion while printing circular structure.

\*print-escape\* \([\text{NIL}]\)  \> If \(\text{NIL}\), do not print escape characters and package prefixes.

\*print-gensym\* \([\text{NIL}]\)  \> If \(T\), print #: before uninterred symbols.

\*print-length\* \([\text{NIL}]\)  \> 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\* \([\text{NIL}]\)  \> If \(T\), print rationals with a radix indicator.

\*print-readably\* \([\text{NIL}]\)  \> If \(T\), print \(\text{readably}\) or signal error \(\text{print-not-readable}\).

\*print-right-margin\* \([\text{NIL}]\)  \> Right margin width in ems while pretty-printing.

\(\text{set-pprint-dispatch} \text{type function} \text{[priorities]} \text{[table]} \) \> Install entry comprising \text{function} arguments stream and object to print; and \text{priority} as \text{type} into \text{table}. If \text{function} is \(\text{NIL}\), remove \text{type} from \text{table}. Return \(\text{NIL}\).

\(\text{pprint-dispatch} \text{foo} \text{[table]} \) \> Return highest \text{priority} \text{function} associated with type of \text{foo} and \(T\) if there was a matching type specifier in \text{table}.

\(\text{copy-pprint-dispatch} \text{[table]} \) \> Return \text{copy} of \text{table} or, if \text{table} is \(\text{NIL}\), \text{initial} value of \text{*print-pprint-dispatch*}.

\*print-pprint-dispatch*  \> Current pretty print dispatch table.

13.5 Format

\(\text{#format control} \) \> Return function of \text{stream} and \text{arg}\(^*\) applying \text{format} to \text{stream}, \text{control}, and \text{arg}\(^*\) returning \text{NIL} or any excess \text{args}.

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

- \[\text{min-co} \] \[\text{[col-in]}\] \[\text{[min-pad]}\] \[\text{[*pad-char]}\] \]  \> Aesthetic/Standard. Print argument of any type for consumption by humans/ by the reader, respectively. With \(\_\), print \text{NIL} as \(\_\) rather than \text{NIL}; with \(\emptyset\), add \text{pad-chars} on the left rather than on the right.

\(\text{nth-value n form} \) \> Zero-indexed \(n\)th return value of \text{form}.

\(\text{complement function} \) \> Return new function with same arguments and same side effects as \text{function}, but with complementary truth value.

\(\text{constantly foo} \) \> \text{Function} of any number of arguments returning \text{foo}.

\(\text{identity foo} \) \> Return \text{foo}.

\(\text{function-lambda-expression function} \) \> If available, return lambda expression of \text{function}. \text{NIL} if \text{function} was defined in an environment without bindings, and \text{name} of \text{function}.

\(\text{definition \{foo \{set foo\}\}} \) \> Definition of global function \text{foo}. \text{setable}.

\(\text{makunbound foo} \) \> Remove global function or macro definition \text{foo}.

\(\text{call-arguments-limit} \) \> Upper bound of the number of function arguments or lambda list parameters, respectively; \(\geq 50\).

\(\text{multiple-values-limit} \) \> Upper bound of the number of values a multiple value can have; \(\geq 20\).

9.4 Macros

Below, macro lambda list \(\text{(macro-\lambda^*\_)}\) has the form of either

\[\text{([&whole var] [E] \{var \text{(macro-\lambda^*)} \{E\}]^{*} \}}\]

\[
\begin{align*}
\text{[&optional} & \{\{\text{var \text{(macro-\lambda^*)}} \{\text{init} \text{[supplied-p]]}\}\} \{E\}]^{*} \\
\text{[&rest} & \{\text{var \text{(macro-\lambda^*)}}\} \{E\}] \\
\text{[&body} & \{\text{var \{tkey \{var \text{(macro-\lambda^*)}\}\} \{\text{init} \text{[supplied-p]]}\}\} \{E\}]^{*} \\
\text{[&allow-other-keys]} & \{\text{var \{var \text{[init]}\}} \{\{E\}\}\} \{E\}] \\
\text{[&aux} & \{\text{var \text{[init]}\}} \{\{E\}\}\} \{E\}] \\
\text{[&optional} & \{\{\text{var \text{(macro-\lambda^*)}} \{\text{init} \text{[supplied-p]]}\}\} \{E\} \{\text{. rest-var}\}.
\end{align*}
\]

One toplevel \(E\) may be replaced by \&environment \text{var}. \text{supplied-p} is \(T\) if there is a corresponding argument. \&init forms can refer to any \text{init} and \text{supplied-p} to their left.

\[\text{edefmacro \{\text{declare \text{decl}^*\}} \{foo \{\text{set foo}\}\} \text{(macro-\lambda^*)}}\]

\[\text{\{doc \}} \{\text{form^*}\}\]  \> Define macro \text{foo} which on evaluation as \text{foo tree} applies expanded forms to arguments from tree, which corresponds to \text{tree-shaped macro-\text{As}}. forms are enclosed in an implicit \text{block} named \text{foo}.
(define-symbol-macro foo form)
  ▶ Define symbol macro foo which on evaluation evaluates expanded form.

(macrolet ((foo (macro-λ*) (declare local-decl*)* macro-form))
  (declare decl*) form)
  ▶ Evaluate forms with locally defined mutually invisible macros foo which are enclosed in implicit .blocks of the same name.

(symbol-macrolet ((foo expansion-form*) (declare decl*)* form)
  ▶ Evaluate forms with locally defined symbol macros foo.

(defun function (arg var) (declare decl*) form)
where defset lambda list (set-λ*) has the form
  (var* (optional var) (var init supplied-p)) \& rest var
\&key (var (when var) init supplied-p) \& allow-other-keys \& environment var
  ▶ Specify how to set a place accessed by function. Short form: (setf function arg) value-form is replaced by (update 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 values accessed where set-λ and s-var* describe the arguments of function and the value(s) to be stored, respectively; and that returns the value(s) of s-var*. forms are enclosed in an implicit .block named function.

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

(get-set-expansion place (environment true))
  ▶ Return lists of temporary variables arg-vars and of corresponding args as given with place, list newval-vars with temporary variables corresponding to the new values, and set-form and get-form specifying in terms of arg-vars and newval-vars how to set and how to read place.

(define-macro macro foo (\optional (var init supplied-p)) \& rest var)
  ▶ 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.

(write-sequence sequence stream)
  ▶ Write elements of sequence to binary or character stream.

  \{start string \}
  ▶ Print foo to stream and return foo, or print foo into string, respectively, after dynamically setting printer variables corresponding to keyword parameters (*print-bar* becoming :bar).

  (stream stream \{standard-output\})
  ▶ Print foo to string. If foo is a list, print as many elements per line as possible; do the same in a table with a column width of n ems; or print either all elements on one line or each on its own line, respectively. Return NIL. Usable with \format directive -f/

  \{write-to-string\} foo
  ▶ Print foo to stream and return foo.

  (write-to-string)
  ▶ Print foo to stream and return foo.

  (write-string string)
  ▶ Print foo to stream and return foo.

  (write-raw string)
  ▶ Print foo to stream and return foo.

  (write-sequence sequence stream)
  ▶ Write elements of sequence to binary or character stream.

  (\write)
  ▶ Write elements of sequence to binary or character stream.

  \{start string \}
  ▶ Write elements of sequence to binary or character stream.

  (\write)
  ▶ Write elements of sequence to binary or character stream.

  (\write)
  ▶ Write elements of sequence to binary or character stream.

  (\write)
  ▶ Write elements of sequence to binary or character stream.

  (\write)
  ▶ Write elements of sequence to binary or character stream.

  (\write)
  ▶ Write elements of sequence to binary or character stream.
Common Lisp Quick Reference

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

#*[type {slot value}]*  ➢ Structure of type.

#*[string*  ➢ A pathname.

#*[foo*  ➢ Uninterned symbol foo.

#*[form*  ➢ Read-time value of form.

.*read-eval*  ➢ If NIL, a reader-error is signalled at #.

.#integer= foo*  ➢ Give foo the label integer.

.#integer #  ➢ Object labelled integer.

.#<  ➢ Have the reader signal reader-error.

#*[feature when-feature*  ➢ Means when-feature if feature is T; means unless-feature if feature is NIL. feature is a symbol from #features, or ({and feature} or {or feature}), or (not feature).

#*[features*  ➢ List of symbols denoting implementation-dependent features.

|c*| \c  ➢ Treat arbitrary character(s) c as alphabetic preserving case.

Common Lisp Quick Reference

13.4 Printer

#*[print#1 [stream standard-output]*  ➢ Print foo to stream readable, readably between a newline and a space, readably after a newline, or human-readable without any extra characters, respectively. #print, #print and #princ return foo.

#*[print1-to-string foo*  ➢ Print foo to string readable or human-readable, respectively.

#*[print-object object stream*  ➢ Print object to stream. Called by the Lisp printer.

#*[print-unreadable-object (foo stream) {type book \[identity \]} form*  ➢ Enclosed in < or >, print foo by means of forms to stream. Return NIL.

#*[terpri [stream standard-output]*  ➢ Output a newline to stream. Return NIL.

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

#*[write-char char [stream standard-output]*  ➢ Output char to stream.

#*[write-string string [stream standard-output] {start start\}}*  ➢ Write string to stream without/with a trailing newline.

#*[write-byte byte [stream]*  ➢ Write byte to binary stream.

9.5 Control Flow

#*[if test then [else]*  ➢ Return values of then if test returns T; return values of else otherwise.

#*[cond (test then* [else]*)*  ➢ Return the values of the first then* whose test returns T; return NIL if all tests return NIL.

#*[when test foo]*  ➢ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.

#*[case (test (key*) [foo*] [key]) (otherwise bar)]*  ➢ Return the values of the first foo* one of whose keys is eql test. Return values of bars if there is no matching key.

#*[case test (key*) foo* [key])*  ➢ Return the values of the first foo* one of whose keys is eql test. Signal non-correctable/correctable type-error if there is no matching key.

#*[and form*]*  ➢ Evaluate forms from left to right. Immediately return NIL if one form’s value is NIL. Return values of last form otherwise.

#*[or form]*  ➢ 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.

#*[progn form*]*  ➢ Evaluate forms sequentially. Return values of last form.

#*[multiple-value-prog prog form]*  ➢ Evaluate forms in order. Return values/primary value, respectively, of form-r.

#*[multiple-value-prog prog form]*  ➢ Evaluate forms in order. Return values/primary value, respectively, of form-r.

#*[prog prog*]*  ➢ Evaluate prog’s name lexically bound (in parallel or sequentially, respectively) to values. Return NIL or explicitly returned values. Implicitly, the whole form is a block named NIL.

#*[unwind-protect protected cleanup]*  ➢ Evaluate protected and then, no matter how control leaves protected, cleanups. Return values of protected.

#*[block name form]*  ➢ Evaluate forms in a lexical environment, and return their values unless interrupted by \return-from.

#*[return-from foo result]*  ➢ Have nearest enclosing block named foo/named NIL, respectively, return with values of result.
9.6 Iteration

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

\[
\begin{align*}
\text{dotimes} & \quad \{ \text{var \ i \ \text{result}} \} \ (\text{declare \ decl}^*) \* \{ \text{tag} \{ \text{form} \} \} \\
& \quad \Rightarrow \text{Evaluate \ tagbody-like \ body \ with \ var \ successively \ bound \ to \ integers \ from 0 \ to \ i - 1. \ Upon \ evaluation \ of \ \text{result}, \ \text{var} \ is \ \text{i}. \ Implicitly, \ the \ whole \ form \ is \ a \ \text{block} \ named \ \text{NIL}.}
\end{align*}
\]

\[
\begin{align*}
\text{adolist} & \quad \{ \text{var \ list \ \text{result}} \} \ (\text{declare \ decl}^*) \* \{ \text{tag} \{ \text{form} \} \} \\
& \quad \Rightarrow \text{Evaluate \ tagbody-like \ body \ with \ var \ successively \ bound \ to \ the \ elements \ of \ list. \ Upon \ evaluation \ of \ \text{result}, \ \text{var} \ is \ \text{NIL}. \ Implicitly, \ the \ whole \ form \ is \ a \ \text{block} \ named \ \text{NIL}.}
\end{align*}
\]

9.7 Loop Facility

\[
\begin{align*}
\text{loop} & \quad \{ \text{form} \} \\
& \quad \Rightarrow \text{Simple Loop. If \ forms \ do \ not \ contain \ any \ atomic \ Loop \ Facility \ keywords, \ evaluate \ them \ forever \ in \ an \ implicit \ \text{block} \ named \ \text{NIL}.}
\end{align*}
\]

\[
\begin{align*}
\text{loop} \ \text{clause} & \quad \Rightarrow \text{Loop Facility. For \ Loop \ Facility \ keywords \ see \ below \ and \ Figure \ 1.}
\end{align*}
\]

named \text{n} \Rightarrow \text{Give \ \text{n}'s \ implicit \ \text{block} \ a \ name.}

\[
\begin{align*}
\{ \text{with} \{ \text{var-s} \{ \text{var-s}^* \} \} \{ \text{d-type} \} \Rightarrow \text{foo} \}^*  \\
\{ \text{and} \{ \text{var-p} \{ \text{var-p}^* \} \} \{ \text{d-type} \} \Rightarrow \text{bar} \}^* \\
\end{align*}
\]

where \text{destructuring \ type \ specifier} \ d-type \ \text{has \ the \ form} \ \{ \text{fixnum} \{ \text{float} \} \{ \text{integer} \} \{ \text{of-type} \{ \text{type} \} \} \} \\
\Rightarrow \text{Initialize \ (possibly \ trees \ of) \ local \ variables \ var-s \ sequentially \ and \ var-p \ in \ parallel.}

\[
\begin{align*}
\{ \text{foras} \{ \text{var-s} \{ \text{var-s}^* \} \} \{ \text{d-type} \} \} \{ \text{and} \{ \text{var-p} \{ \text{var-p}^* \} \} \{ \text{d-type} \} \} \\
\end{align*}
\]

\Rightarrow \text{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} \ \text{with.}
13.2 Reader

\begin{itemize}
\item \{y-or-n-p\} \hspace{1cm} \text{(control arg*)}
\item \{yes-or-no-p\}
\hspace{1cm} \text{Ask user a question and return T or NIL depending on their answer. See page 38, \texttt{/format}, for control and args.}
\item (\texttt{with-standard-to-syntax form})
\hspace{1cm} \text{Evaluate forms with standard behaviour of reader and printer. Return values of forms.}
\item \text{read}
\hspace{1cm} \text{(read-preserving-whitespace \{stream \text{standard-input} \[eof-error \[eof-val \ Recursive])\}})
\hspace{1cm} \text{Read printed representation of object.}
\item \text{read-from-string string \{eof-error \[eof-val \ Recursive])\}}
\hspace{1cm} \text{Return object read from string and zero-indexed position of next character.}
\item \text{read-delimited-list char \{stream \text{standard-input} \[recursive \ Recursive])\}}
\hspace{1cm} \text{Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.}
\item \text{read-char \{stream \text{standard-input} \[eof-error \[eof-val \ Recursive])\}}
\hspace{1cm} \text{Return next character from stream.}
\item \text{read-char-no-hang \{stream \text{standard-input} \[eof-error \[eof-val \ Recursive])\}}
\hspace{1cm} \text{Next character from stream or NIL if none is available.}
\item \text{peek-char \{mode \text{standard-input} \[eof-error \[eof-val \ Recursive])\}}
\hspace{1cm} \text{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.}
\item \text{unread-char character \{stream \text{standard-input})\}}
\hspace{1cm} \text{Put last read-charred character back into stream; return NIL.}
\item \text{read-byte \{stream \[eof-error \[eof-val \ Recursive])\}}
\hspace{1cm} \text{Read next byte from binary stream.}
\item \text{read-line \{stream \[eof-error \[eof-val \ Recursive])\}}
\hspace{1cm} \text{Return a line of text from stream and T if line has been ended by end of file.}
\item \text{read-sequence sequence \{stream \[start \[end \Recursive])\}}
\hspace{1cm} \text{Replace elements of sequence between start and end with elements from binary or character stream. Return index of sequence's first unmodified element.}
\item \text{readtable-case readtable \{stream})\}}
\hspace{1cm} \text{Case sensitivity attribute (one of :upcase, :downcase, \texttt{:preserve}, \texttt{:invert}) of readtable, settable.}
\item \text{copy-readtable \{from-readtable \[to-readtable \Recursive])\}}
\hspace{1cm} \text{Return copy of from-readtable.}
\item \text{set-syntax-from-char to-char from-char \[to-readtable \Recursive])\}}
\hspace{1cm} \text{Copy syntax of from-char to to-readtable. Return T.}
\end{itemize}

\texttt{*readtable*} \hspace{1cm} \text{Current readtable.}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{loop_facility.png}
\caption{Loop Facility, Overview.}
\end{figure}
(upfrom|from|downfrom) start
▷ Start stepping with start.
(upto| downto| below| above) form
▷ Specify form as the end value for stepping.
(on|in) list
▷ Bind var to successive elements/tails, respectively, of list.
by (step|function|rcons):
▷ Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.
= foo [then bar|else]
▷ 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 termination.
{append|appending|nconc|nconc} {form|it} into list
▷ Concatenate values of form or it, which should be lists, into list by the means of ,append or ,nconc, respectively. If no list is given, collect into an anonymous list which is returned after termination.
{count|counting} {form|it} into n [type]
▷ Count the number of times the value of form or of it is T. If no n is given, count into an anonymous variable which is returned after termination.
{sum|summing} {form|it} into sum [type]
▷ Calculate the sum of the primary values of form or of it. If no sum is given, sum into an anonymous variable which is returned after termination.
{maximize|maximizing|minimize|minimizing} {form|it} into max-min [type]
▷ Determine the maximum or minimum, respectively, of the primary values of form or of it. If no max-min is given, use an anonymous variable which is returned after termination.

13 Input/Output

13.1 Predicates

(stream foo)
▷ ★ if foo is of indicated type.
(pathnamem pfoo)  ▷ T if path matches wildcard.
(readtablep foo)

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

(pathname-match-p path wildcard)
▷ T if path matches wildcard.
(wild-pathname-p path [{:host|device|directory|name|type|version|NIL}])
▷ Return T if indicated component in path is wildcard. (NIL indicates any component.)
{initially} \text{finally} \text{form}^+ \rightarrow \text{Evaluate forms before begin, or after end, respectively, of iterations.}

\text{repeat} \text{num} \rightarrow \text{Terminate } \text{loop} \text{ after num iterations; num is evaluated once.}

\{\text{while/untily} \text{test} \rightarrow \text{Continue iteration until test returns NIL or T, respectively.}

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

\text{thereis} \text{test} \rightarrow \text{Terminate } \text{loop} \text{ when test is T and return value of test, skipping any finally parts. Otherwise continue } \text{loop} \text{ with its default return value set to NIL.}

\text{(mloop-finish)} \rightarrow \text{Terminate } \text{loop} \text{ immediately executing any finally clauses and returning any accumulated results.}

10 CLOS

10.1 Classes

\text{\texttt{(slot-exists-p foo bar)}} \rightarrow \text{T if foo has a slot bar.}

\text{\texttt{(slot-boundp instance slot)}} \rightarrow \text{T if slot in instance is bound.}

\text{\texttt{(defclass foo \texttt{(superclass\texttt{*}) standard-object)}}}

\text{\texttt{name value \ldots}} \rightarrow \text{Define or modify class foo as a subclass of superclasses. Transform existing instances, if any by \texttt{make-instances-obsolete}. In a new instance i of foo, a slot’s value defaults to form unless set via \texttt{|initarg-name|}; it is readable via \texttt{(reader i)} or \texttt{(accessor i)}, and writable via \texttt{(writer value i)} or \texttt{(setf accessor i value)}. slots with \texttt{allocation :class} are shared by all instances of class foo.}

\text{\texttt{(find-class symbol [error \texttt{(environment)}])}} \rightarrow \text{Return class named symbol. setfable.}

\text{\texttt{(make-instance class \texttt{(|initarg value|) other-keyarg*)}} \rightarrow \text{Make new instance of class.}

\text{\texttt{(reinitialize-instance instance \texttt{(|initarg value|) other-keyarg*)}} \rightarrow \text{Change local slots of instance according to initargs by means of shared-initialize.}

\text{\texttt{(slot-value foo slot)}} \rightarrow \text{Return value of slot in foo, setfable.}

\text{\texttt{(slot-makunbound instance slot)}} \rightarrow \text{Make slot in instance unbound.}
10.2 Generic Functions

(include-p (slot (var slot)) instance (declare decl) form)
> Return values of forms after evaluating them in a lexical environment with slots of instance visible as settable slots or \texttt{vars}/with accessors of instance visible as settable vars.

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

(update-instance-for-redefined-class)
> Class \texttt{foo} is a direct instance of.

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

(make-instances-obsolete class)
> Update all existing instances of \texttt{class} using \texttt{update-instance-for-redefined-class}.

(standard-method instance)
> Return uninitialized instance of \texttt{class}. Called by \texttt{make-instance}.

(shared-initialize instance (\texttt{initform-slots}) {([\texttt{initarg} value]} other-keyarg*)
> Fill the \texttt{initarg}-slots of instance with the corresponding values, and fill those \texttt{initform-slots} that are not \texttt{initarg}-slots with the values of their :\texttt{initform} forms.

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

12 Types and Classes

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

(typep foo type [environment])
> T if \texttt{foo} is of type.

(subtypep type-a type-b [environment])
> Return T if type-a is a recognizable subtype of type-b, and NIL if the relationship could not be determined.

(the type form)
> Declare values of form to be of type.

(coerce object type)
> Coerce object into type.

(typecase form (\texttt{a-form})* (otherwise b-forms)
> Return values of the first \texttt{a-form} whose type is \texttt{foo} of. Return values of \texttt{b-forms} if no type matches.

(metatypecase form (\texttt{type form})
> Return values of the first \texttt{form} whose type is \texttt{foo} of. Signal non-convertable/convertable \texttt{type-error} if no type matches.
\begin{verbatim}
(handler-case foo (type (var) (declare decl*)* condition-form*)
  ([no-error (ord-λ*) (declare decl*)* form*)])
  \triangleright 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-λ 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*)
  \triangleright Return values of forms after evaluating them with
  condition-types dynamically bound to their respective
  handler-functions of argument condition.

(with-simple-restart (restart form*) control arg*)
  \triangleright 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.

(restart-case form (restart (ord-λ*))
  \triangleright 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*.

(restart-bind ((``restart NIL) restart-function
  \triangleright 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.

(invoke-restart restart arg*)

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

(find-restart name) [condition]
  \triangleright Return innermost restart name, or a list of all restarts, re-
  spectively, out of those either associated with condition or un-
  associated at all; or, without condition, out of all restarts. Re-
  turn NIL if search is unsuccessful.

(restart-name restart)
  \triangleright Name of restart.

(abort muffle-warning) [condition\emph{nil}]

(continue store-value value [condition\emph{nil}])

(use-value value)
\end{verbatim}
10.3 Method Combination Types

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

\[ \text{method} \]
- Simple built-in method-combination types: have the same usage as the \( \text{c-type} \)s defined by the short form of \( \text{method-combination} \).

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

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

\[ \text{call-method} \]
- From within an effective method form, call \( \text{method} \) 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.