Contents

1 Numbers 3 9.5 Control Flow . . . . 19
   1.1 Predicates . . . . 3 9.6 Iteration . . . . 21
   1.2 Numeric Functns . . . 3 9.7 Loop Facility . . . . 21
   1.3 Logic Functions . . . 4 10 CLOS 24
   1.4 Integer Functions . . . 5 10.1 Classes . . . . 24
   1.5 Implementation-Dependent . . . 6 10.2 Generic Functns . 25
   10.3 Method Combina-
   tion Types . . . . 26
2 Characters 6
3 Strings 7
4 Conses 8
   4.1 Predicates . . . . 8
   4.2 Lists . . . . . . 8
   4.3 Association Lists . . . 9
   4.4 Trees . . . . . . 10
   4.5 Sets . . . . . . 10
5 Arrays 10
   5.1 Predicates . . . . 10
   5.2 Array Functions . . . 10
   5.3 Vector Functions . . . 11
6 Sequences 12
   6.1 Seq. Predicates . . . 12
   6.2 Seq. Functions . . . 12
7 Hash Tables 14
8 Structures 15
9 Control Structure 15
   9.1 Predicates . . . . 15
   9.2 Variables . . . . 16
   9.3 Functions . . . . 17
   9.4 Macros . . . . . 18
10 CLOS 24
   10.1 Classes . . . . . 24
   10.2 Generic Functns . 25
   10.3 Method Combina-
   tion Types . . . . 26
11 Conditions and Errors 27
12 Types and Classes 30
13 Input/Output 32
   13.1 Predicates . . . . 32
   13.2 Reader . . . . . 32
   13.3 Character Syntax . 33
   13.4 Printer . . . . . 34
   13.5 Format . . . . . 36
   13.6 Streams . . . . . 39
   13.7 Paths and Files . . 40
14 Packages and Symbols 42
   14.1 Predicates . . . . 42
   14.2 Packages . . . . 42
   14.3 Symbols . . . . . 43
15 Compiler 44
   15.1 Predicates . . . . 44
   15.2 Compilation . . . . 44
   15.3 REPL & Debug . . 45
   15.4 Declarations . . . . 46
16 External Environment 47

Typographic Conventions

name; name; name; name; name; name; 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] ▶ 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}; {foo|bar|baz}; {foo|bar|baz};
   ▶ Either foo, or bar, or baz.

\{foo|bar|baz\}; \{foo|bar|baz\}; \{foo|bar|baz\}; \{foo|bar|baz\};
   ▶ Anything from none to each of foo, bar, and baz.

foo ▶ Argument foo is not evaluated.

bar ▶ Argument bar is possibly modified.

foo* ▶ foo* is evaluated as in \progn; see page 20.

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

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

1.1 Predicates

\( (= \mathbf{num}) \)  
\( (= \mathbf{number}) \)  
\( (= \mathbf{symbol}) \)  
\( (= \mathbf{number} \mathbf{?}) \)  
\( (= \mathbf{symbol} \mathbf{?}) \)  
\( (= \mathbf{num} \mathbf{?}) \)  

\( \Rightarrow \) if all numbers, or none, respectively, are equal in value.
\( \Rightarrow \) if any number is NaN, floating point non-monotonic, or non-monotonic and non-decreasing, respectively.

\( \text{minusp \ a} \)  
\( \text{zerop \ a} \)  
\( \text{plusp \ a} \)  
\( \text{evenp \ int} \)  
\( \text{oddp \ int} \)  
\( \text{numberp \ foo} \)  
\( \text{realp \ foo} \)  
\( \text{rationalp \ foo} \)  
\( \text{floatp \ foo} \)  
\( \text{integerp \ foo} \)  
\( \text{complexp \ foo} \)  
\( \text{random-state-p \ foo} \)  

1.2 Numeric Functions

\( \text{frac} \ a \)  
\( \text{frac} \ b \)  
\( \text{gcd} \ a \)  

\( \Rightarrow \) Return \( + a \) or \( -a \), respectively.
\( \Rightarrow \) Increment or decrement the value of place by delta. Return new value.
\( \Rightarrow \) Return \( a^{b} \) or \( b^{a} \), respectively.
\( \Rightarrow \) Return log_a(b) or, without b, \( \ln a \).

\( \text{sqrt} \ n \)  
\( \text{sqrt} \ n \)  

\( \Rightarrow \) in complex numbers/natural numbers.

\( \text{log10} \)  
\( \text{logb} \)  

\( \Rightarrow \) Least common multiple or greatest common denominator, respectively, of integers. \( \gcd \) returns 0.

\( \text{remainder} \)  
\( \text{remainder} \)  

\( \Rightarrow \) long-float approximation of \( \pi \), Ludolph's number.

\( \text{sin} \ a \)  
\( \text{cos} \ a \)  
\( \text{tan} \ a \)  

\( \Rightarrow \) sin a or cos a, respectively. (a in radians.)
\( \Rightarrow \) arccos a or arccos a, respectively, in radians.

\( \text{atan} \ a \)  
\( \text{atan2} \)  

\( \Rightarrow \) arctan a or \( \pm \arctan a \), respectively, in radians.

\( \text{sinh} \ a \)  
\( \text{cosh} \ a \)  
\( \text{tanh} \ a \)  

\( \Rightarrow \) sinh a, cosh a, or tanh a, respectively.
\[
\begin{align*}
\text{asinh} a & \mapsto \sinh^{-1} a, \quad \text{acosh} a \text{ or } \text{atanh} a, \\
\text{atanh} a & \mapsto \text{atanh} a, \\
\text{cis} a & \mapsto \cos a + i \sin a, \\
\text{conjugate} a & \mapsto \text{Return complex conjugate of } a, \\
\max \text{ num } & \mapsto \text{Greatest or least, respectively, of numns}, \\
\min \text{ num } & \mapsto \text{Result of magnitude 1 representing sign or phase of } n, \\
\text{fLOOR, floor} & \mapsto \text{Return as integer or float, respectively, n/d rounded, or}
\quad \text{rounded towards } -\infty, +\infty, \text{or 0, respectively; and remain-}
\quad \text{der.} \\
\text{random-limit \{state\}} & \mapsto \text{Return non-negative \textbf{random} number less than \textbf{limit} and, of the same type.} \\
\text{random-state} & \mapsto \text{Current \textbf{random} state.} \\
\text{FLOAT-SIGN} & \mapsto \text{Number of magnitude 1 representing sign or phase of } n. \\
\text{Numerator rational} & \mapsto \text{Denominator rational}, \quad \text{or of } n, \text{of } \textbf{rational's} \text{ canonical form.} \\
\text{realpart number} & \mapsto \text{Real \textbf{part} or imaginary \textbf{part}, respectively, of number.} \\
\text{imagpart number} & \mapsto \text{Return } n. \\
\text{complex real image} & \mapsto \text{Make a complex \textbf{number}.} \\
\text{phase num} & \mapsto \text{Angle of num's \textbf{polar} representation.} \\
\text{abs} n & \mapsto \text{Return } |n|. \\
\text{rational real} & \mapsto \text{Return } n. \\
\text{rationalize real} & \mapsto \text{Convert real to rational. Assume complete/limited accuracy for real.} \\
\text{float real \{prototype\}} & \mapsto \text{Convert real into float with type of prototype.} \\
\end{align*}
\]

### 1.3 Logic Functions

Negative integers are used in two different complement representations.

\[
\begin{align*}
\text{boole operation int-a int-b} & \mapsto \text{Return value of bitwise logical operation.} \\
\text{boole-1} & \mapsto \text{int-a}, \\
\text{boole-2} & \mapsto \text{int-b}, \\
\text{boole-c1} & \mapsto \text{int-a,} \\
\text{boole-c2} & \mapsto \text{int-b,} \\
\text{boole-set} & \mapsto \text{All bits set,} \quad \text{boole-clr} \mapsto \text{All bits zero.} \\
\end{align*}
\]
1.5 Implementation-Dependent

- short-float
- single-float
- double-float
- long-float

- least-negative
- least-negative-normalized
- least-positive
- least-positive-normalized

- most-negative
- most-positive

- short-float
- single-float
- double-float
- long-float

- fixnum

- type

- implementation-dependent

- implementation-defined

- implementation-specific

- standard-character

- character

- integer

- character-normalized

- character-p

- character-equal

- character-not-equal

- character-less

- character-less-or-equal

- character-greater

- character-greater-or-equal

- character-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

- float-digit-less

- float-digit-less-or-equal

- float-digit-greater

- float-digit-greater-or-equal

- float-digit-ordered

- float

- floating-point

- floating-point-normalized

- floating-point-normalized-with-type-p

- floating-point-un-normalized

- floating-point-un-normalized-with-type-p

- float-equality

- float-precision

- float-precision-with-type-p

- float-precision-p

- float-precision-not-p

- float-precision-less

- float-precision-less-or-equal

- float-precision-greater

- float-precision-greater-or-equal

- float-precision-ordered

- float-digit

- float-digit-with-type-p

- float-digit-p

- float-digit-not-p

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

- `stringp x`  
  > T if `x` is of indicated type.

- `string= x y`  
  > T if subsequences of `x` and `y` are equal. Otherwise return `NIL`. Obey/ignore, respectively, case.

- `string x`  
  > Convert `x` into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

- `string-capitalize string`  
  > Convert `string` into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

- `string-ucase string`  
  > Convert `string` into an all-lowercase string.

- `string-lcase string`  
  > Convert `string` into an all-lowercase string.

- `string-trim string`  
  > Return string with all characters in sequence `char-bag` removed from both ends, from the beginning, or from the end, respectively.
4 Conses

4.1 Predicates

(consfoo) → Return T if foo is of indicated type.
(listfoo) → Return T if list/fooo is NIL.
(endp) → Return T if foo is not a cons.
(atomfoo) → Return T if foo is a tail of list.
(memberfoo) → Return T if list starting with its first element matching foo. Return NIL if there is no such element.
(member-if) → Return tail of list starting with its first element satisfying test. Return NIL if there is no such element.
(subsetp) → Return T if list-a is a subset of list-b.

4.2 Lists

(consfoo bar) → Return new cons (foo, bar).
(listfoo) → Return list of foos.
(listfoo) → Return list of foos with last foo becoming cdr of last cons. Return foo if only one foo given.
(make-list num initial-element) → New list with num elements set to foo.
(list-length) → Length of list; NIL for circular list.
(car) → Car of list or NIL if list is NIL. setfable.
(cdr) → Cdr of list or NIL if list is NIL. setfable.
(rest) → Return tail of list after calling cdr n times.
(nthcdr) → Return nth element of list if any, or NIL otherwise. setfable.
(nth) → Zero-indexed nth element of list. setfable.
(cadrlist) → With X being one to four as and ds representing cars and cdrs, e.g. (cadr bar) is equivalent to (car (cdr bar)). setfable.
(last list num) → Return list of last num conses of list.
(symbol-name symbol)
(symbol-package symbol)
(symbol-plist symbol)
(symbol-value symbol)
(symbol-function symbol)
   ▶ Name, package, property list, value, or function, respectively, of symbol. 
(setf documentation new-doc)
   ▶ Get/set documentation string of doc of given type.

NIL() ▶ Truth; the supertype of every type including t; the superclass of every class except t, *terminal-io*.
nil() ▶ Falsy; the empty list; the empty type, subtype of every type; *standard-input*; *standard-outputs*; the global environment.

14.4 Standard Packages

common-lispcl
   ▶ Exports the defined names of Common Lisp except for those in the keyword package.
common-lisp-user-cl-user
   ▶ Current package after startup; uses package common-lisp.
keyword
   ▶ Contains symbols which are defined to be of type keyword.

15 Compiler

15.1 Predicates

(special-operator-p foo) ▶ T if foo is a special operator.
(compiled-function-p foo) ▶ T if foo is of type compiled-function.

15.2 Compilation

(compile
   {NIL definition
      {setf name} [definition]})
   ▶ Return compiled function or replace name’s function definition with the compiled function. Return T in case of warnings or errors, and T in case of warnings or errors excluding style-warnings.

(output-file out-path)
   ▶ Write compiled contents of file to out-path. Return success or NIL, T in case of warnings or errors, T in case of warnings or errors excluding style-warnings.

(compile-file file)
   ▶ Pathname. compile-file writes to if invoked with the same arguments.

(load path)
   ▶ Load source file or compiled file into Lisp environment. Return T if successful.

4.3 Association Lists

(pairlis keys values [alist head])
   ▶ Prepends to alist an association list made from lists keys and values.

(acons key-value alist)
   ▶ Return alist with a (key, value) pair added.

(assoc list)
   ▶ First cons whose car, or cdr, respectively, satisfies test.

(copy-alist alist) ▶ Return copy of alist.
4.4 Trees

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

(subst new old tree)
   - Make copy of tree with each subtree or leaf matching old replaced by new.
   - Return T if trees foo and bar have same shape and leaves.

(subst-if-not new old tree)
   - Copy tree with each subtree or leaf satisfying test replaced by new.

(subsub association-list tree)
   - 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.

5 Arrays

5.1 Predicates

(arrayp foo)
   - T if foo is of indicated type.

(vector foo)

(simple-vector-p foo)

(bit-vector-p foo)

(simple-bit-vector-p foo)

(adjointable-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

(make-array dimension-sizes [adjustable [active]])

(adjoint-array dimension-sizes)
   - Return a T if foo is a symbol.

(find-symbols assoc-list)
   - T if foo is a symbol.

(import symbol [package [package]]
   - T if foo is a symbol.

(shadow symbols [package]
   - T if foo is a symbol.

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

(export symbols [package [package]]
   - T if foo is a symbol.

(unexport symbols [package [package]]
   - T if foo is a symbol.

(do-symbols [do-external-symbols]
   - T if foo is a symbol.

(with-package-[package]
   - T if foo is a symbol.

(require module [path])
   - T if foo is a symbol.

(provide module)
   - T if foo is a symbol.

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)
   - T if foo is a symbol.

(gensym)
   - T if foo is a symbol.

(gentemp [prefix [package]]
   - T if foo is a symbol.

(copy-symbol symbol [props]
   - T if foo is a symbol.


14 Packages and Symbols

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

14.1 Predicates

(symbolp foo)  ▷ ▷ If foo is of indicated type.

14.2 Packages

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

package:symbol  ▷ ▷ Exported symbol of package.

package:symbol  ▷ ▷ Possibly exported symbol of package.

(depackage foo)

: (nicknames nick*)

: (intern intern-symbol*)

: (use used-package*)

: (import-from pkg imported-symbol*)

: (shadowing-import-from pkg shd-symb*)

: (shadow shd-symbol*)

: (export exported-symbol*)

:size int

▷ ▷ Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and shd-symbols. Add shd-symbols to foo’s shadowing list.

(make-package foo)

: (nicknames (nick*)

: (use (used-package*))

▷ ▷ Create package foo.

(rename-package package new-name [new-nicknames])

▷ ▷ Rename package. Return renamed package.

(in-package foo)

▷ ▷ Make package foo current.

(use-package other-packages [package package])

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

(package-use-list package)

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

(delete-package package)

▷ ▷ Delete package. Return T if successful.

*packages

▷ ▷ The current packages.

(list-all-packages)

▷ ▷ List of registered packages.

(package-name package)

▷ ▷ Name of package.

(package-nicknames package)

▷ ▷ Nicknames of package.

(find-package name)

▷ ▷ Package with name (case-sensitive).

(find-all-symbols foo)

▷ ▷ List of symbols foo from all registered packages.

(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])

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

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

▷ ▷ 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.

(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.

(vector foo*)

▷ ▷ Return fresh simple vector of foos.

(svref vector i)

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

(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.

(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 ≥ num if necessary.

(vector-pop vector)

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

(fill-pointer vector)

▷ ▷ Fill pointer of vector. setf-able.
6 Sequences

6.1 Sequence Predicates

\{\texttt{every\,not\,not\,except\,not\,not\,not}\} test sequence\"+

\(\rightarrow\) Return \texttt{NIL} or \(\top\), respectively, as soon as \texttt{test} on any set of corresponding elements of \texttt{sequences} returns \texttt{NIL}.

\{\texttt{some\,
not\,
any\} test sequence\"+

\(\rightarrow\) Return value of \texttt{test} or \texttt{NIL}, respectively, as soon as \texttt{test} on any set of corresponding elements of \texttt{sequences} returns non-\texttt{NIL}.

\{\texttt{mismatch sequence-a sequence-b\}}

\(\rightarrow\) Return position in \texttt{sequence-a} where \texttt{sequence-a} and \texttt{sequence-b} begin to mismatch. Return \texttt{NIL} if they match entirely.

6.2 Sequence Functions

\{\texttt{make-sequence sequence-type size [initial-element]}\}

\(\rightarrow\) Make sequence of \texttt{sequence-type} with size elements.

\{\texttt{concatenate type sequence\}}

\(\rightarrow\) Return concatenated sequence of \texttt{type}.

\{\texttt{merge type sequence-a sequence-b test [key function\}}

\(\rightarrow\) Return interleaved sequence of \texttt{type}. Merged sequence will be sorted if both \texttt{sequence-a} and \texttt{sequence-b} are sorted.

\{\texttt{fill sequence foo \{\texttt{start end}\}}\}

\(\rightarrow\) Return sequence after setting elements between \texttt{start} and \texttt{end} to \texttt{foo}.

\{\texttt{length sequence\}}

\(\rightarrow\) Return length of \texttt{sequence} (being value of fill pointer if applicable).

\{\texttt{count foo sequence \{\texttt{start end}\}}\}

\(\rightarrow\) Return number of elements in \texttt{sequence} which match \texttt{foo}.

\{\texttt{count-if\,
not\,
not\,not\,not\,not\} test sequence \{\texttt{start end}\}}\}

\(\rightarrow\) Return number of elements in \texttt{sequence} which satisfy \texttt{test}.

\{\texttt{elt sequence index\}}

\(\rightarrow\) Return element of \texttt{sequence} pointed to by zero-indexed \texttt{index}. \texttt{setfable}.

\{\texttt{subseq sequence start [end]}\}

\(\rightarrow\) Return subsequence of \texttt{sequence} between \texttt{start} and \texttt{end}. \texttt{setfable}.

\{\texttt{sort sequence [key function\}}

\(\rightarrow\) Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

\{\texttt{reverse sequence\}}

\(\rightarrow\) Return \texttt{sequence} in reverse order.

\{\texttt{reverse sequence\}}

\(\rightarrow\) Return \texttt{sequence} in reverse order.
(close stream [abort bool])
  ▶ Close stream. Return T if stream had been open. If :abort
  is T, delete associated file.

(with-open-file (stream path open-args) [declare decl]* form*)
  ▶ Use :open with open-args to temporarily create stream
  to path; return values of forms.

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

(with-input-from-string (foo string) {index index} [start start\tilde] [declare decl]* form*)
  ▶ Evaluate forms with foo locally bound to input
  string-stream from string. Return values of forms; store
  next reading position into index.

(with-output-to-string (foo string) [element-type
  \ponential type] {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.

(stream-external-format stream)
  ▶ External file format designator.

*terminal-io*  ▶ Bidirectional stream to user terminal.
*standard-input*  ▶ Standard input stream, standard output stream, or stan-
*standard-output*  ▶ dard error output stream, respectively.
*error-output*  ▶ Bidirectional streams for debugging and user interac-
*debug-io*  ▶ tion.
*query-io*  ▶ Bidirectional streams for debugging and user interac-

13.7 Pathnames and Files

(makepathname [host {host NIL unspecific}
  \ condol device {device NIL unspecific}
  \ condol directory \{directory wild[NIL unspecific]
  \ condol \{absolute \relative
  \condol name {file-name wild[NIL unspecific] 
  \condol type \{file-type wild[NIL unspecific] 
  \condol version \{newest version wild[NIL unspecific] 
  \condol defaults path
  \condol \case local common \es/autoloadname]
  \condol]
  ▶ Construct a logical pathname if there is a logical path-
  name translation for host, otherwise construct a physical
  pathname. For :case :local, leave case of components un-
  changed. For :case :common, leave mixed-case components
  unchanged; convert all-uppercase components into local
  customary case; do the opposite with all-lowercase com-
 ponents.

(pathname-host
  \condol pathname-device
  \condol pathname-directory \{case {local \common \es/autoloadname]
  \condol pathname-name
  \condol pathname-type
  \condol pathname-version path-or-stream}
  ▶ Return pathname component.

(find position) foo sequence
  \condol} start \{end \key function
  ▶ Return first element in sequence which matches foo, or
  its position relative to the begin of sequence, respectively.

(find-if \find-if-not \position-if \position-if-not
  \condol test sequence
  \condol} start \{end \key function
  ▶ Return first element in sequence which satisfies test, or
  its position relative to the begin of sequence, respectively.

(search sequence-a sequence-b)
  \condol} start1 \{start2 \end \key function
  \condol\count count
  ▶ Search sequence-b for a subsequence matching sequence-a. Return
  position in sequence-b, or NIL.

(remove foo sequence)
  \condol delete foo sequence
  \condol} start \{end \key function
  \condol\count count
  ▶ Make copy of sequence without elements matching foo.

(remove-if \remove-if-not \delete-if \delete-if-not
  \condol test sequence
  \condol} start \{end \key function
  \condol\count count
  ▶ Make copy of sequence with all (or count) elements sat-
  isfying test removed.

(remove-duplicates sequence)
  \condol delete-duplicates sequence
  \condol} start \{end \key function
  \condol\count count
  ▶ Make copy of sequence without duplicates.

(substitute new old sequence)
  \condol substitute old sequence
  \condol} start \{end \key function
  \condol\count count
  ▶ Make copy of sequence with all (or count) olds replaced
  by new.

(substitute-if \substitute-if-not \ns SUBSTITUTE new old sequence
  \condol substitute-if-not
  \condol new test sequence
  \condol} start \{end \key function
  \condol\count count
  ▶ Make copy of sequence with all (or count) elements sat-
  isfying test replaced by new.
7 Hash Tables

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

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

(replace sequence-a sequence-b)
   ▶ Replace elements of sequence-a with elements of sequence-b.

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

(map-into result-sequence function sequence+)
   ▶ Store into result-sequence successively values of function applied to corresponding elements of the sequences.

(reduce function sequence)
   ▶ Starting with the first two elements of sequence, apply function successively to its last return value together with the next element of sequence. Return last value of function.

(copy-seq sequence)
   ▶ Copy of sequence with shared elements.

13.6 Streams

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

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

(synonym-stream-symbol synonym-stream)
   ▶ Return symbol of synonym-stream.

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

(file-position stream)
   ▶ Return position within stream, or set it to position and return T on success.

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

(listen streams)
   ▶ T if there is a character in input stream.

(clear-input stream)
   ▶ Clear input from stream, return NIL.

(clear-output stream)
   ▶ End output to stream and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

Common Lisp Quick Reference

Common Lisp Quick Reference
8 Structures

(defun defstruct (foo)
  (conc-name foo)
  (conc-name (slot-prefix foo))
  (constructor (make-structure (ord-λ-alist)))
  (copier (copy-structure foo)
    (make-structure (ord-λ-alist)))
  (include-struct (slot (slot init :type slot-type)
    :read-only b)
    (named (initial-offset a))
    (tuple)
    (predicate (p-named function))

  (predict (slot (slot init :type slot-type)
    :read-only b))

  (defun foo (bar)
    (if foo and bar are identical.
      foo)

  (defun eq (foo bar)
    (if foo and bar are identical, or the same character, or numbers of the same type and value.
      foo)

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

  (defun equalp (foo bar)
    (if foo and bar are ?equal? 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.
      foo)

  (defun not (foo)
    (if foo is NIL; NIL otherwise.
      foo)

  (defun boundp (symbol)
    (if symbol is a special variable.
      symbol)

  (defun constantp (foo environment)
    (if foo is a constant form.
      foo)

  (defun functionp (foo)
    (if foo is of type function.
      foo))

9 Control Structure

9.1 Predicates

(defun \# foo bar)
  (if foo and bar are identical.
    foo)

(defun \# eq foo bar)
  (if foo and bar are identical, or the same character, or numbers of the same type and value.
    foo)

(defun \# equal foo bar)
  (if foo and bar are ?equal? 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.
    foo)

(defun \# not foo)
  (if foo is NIL; NIL otherwise.
    foo)

(defun \# boundp symbol)
  (if symbol is a special variable.
    symbol)

(defun \# constantp foo environment)
  (if foo is a constant form.
    foo)

(defun \# functionp foo)
  (if foo is of type function.
    foo)
9.2 Variables

(defconstant foo form (doc))
⇒ Assign value of form to global constant/dynamic variable foo.

(defparameter foo form (doc))
⇒ Unless bound already, assign value of form to dynamic variable foo.

(setf place form)
⇒ Set places to primary values of forms. Return values of last form/NIL; work sequentially/in parallel, respectively.

(set symbol form)
⇒ Set symbols to primary values of forms. Return value of last form/NIL; work sequentially/in parallel, respectively.

(setf place form)
⇒ Set symbol’s value cell to form. Deprecated.

(multiple-value-setq vars form)
⇒ Set elements of vars to the values of form. Return form’s primary value.

(shift place+ foo)
⇒ Store value of foo in rightmost place shifting values of places left, returning first place.

(rotatel place+)
⇒ Rotate values of places left, old first becoming new last place’s value. Return NIL.

(makunbound foo)
⇒ Delete special variable foo if any.

(get symbol key (default NIL))
⇒ First entry key from property list stored in symbol/in place, respectively, or default if there is no key. Settable.

(getf place key (default ()))
⇒ Return key and value of first entry from property-list matching a key from keys, and tail of property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.

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

(remf place key)
⇒ Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.

(let (name (value form))
  (declare (cons* form))
⇒ Evaluate forms with names lexically bound (in parallel or sequentially, respectively) to values. Return values of forms.

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

- \[ \text{min-col} \]
  \[ \text{col-min} \]
  \[ \text{min-pad} \]
  \[ \text{pad-char} \]
  \[ \text{col-interp} \]
  \[ \text{IS} \]
  \[ \text{AS} \]
⇒ Aesthetic/Standard. Print argument of any type for consumption by humans/by the reader, respectively. With \text{col}, print \text{NIL} as \text{0} rather than nil; with \text{AS}, add \text{pad-char} on the left rather than on the right.

- \[ \text{width} \]
  \[ \text{pad-char} \]
  \[ \text{comma-char} \]
  \[ \text{comma-interp} \]
  \[ \text{R} \]
  \[ \text{E} \]
⇒ Radix. (With one or more prefix arguments.) Print argument as number; with \text{R}, group digits \text{comma-interval} each; with \text{E}, always prepend a sign.

- \[ \text{width} \]
  \[ \text{dec-digits} \]
  \[ \text{shift} \]
  \[ \text{overflow-char} \]
  \[ \text{pad-char} \]
  \[ \text{F} \]
⇒ Fixed-Format Floating-Point. With \text{F}, always prepend a sign.

- \[ \text{width} \]
  \[ \text{dec-digits} \]
  \[ \text{exp-digits} \]
  \[ \text{scale-factor} \]
  \[ \text{overlap-char} \]
  \[ \text{pad-char} \]
  \[ \text{E} \]
  \[ \text{G} \]
⇒ Exponential/General Floating-Point. Print argument as floating-point number with \text{dec-digits} after decimal point and \text{exp-digits} in the signed exponent. With \text{G}, choose either \text{E} or \text{F}. With \text{E}, always prepend a sign.

- \[ \text{dec-digits} \]
  \[ \text{int-digits} \]
  \[ \text{width} \]
  \[ \text{pad-char} \]
  \[ \text{S} \]
⇒ Monetary Floating-Point. Print argument as fixed-format floating-point number. With \text{S}, put sign before any padding; with \text{E}, always prepend a sign.

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

- \[ \text{Case-Conversion} \]
  \[ \text{To-Lowercase} \]
  \[ \text{To-Uppercase} \]
⇒ 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.

- \[ \text{Plural} \]
  \[ \text{Eqn} \]
  \[ \text{Eqn-L} \]
⇒ Plural. If argument \text{eqn} print nothing, otherwise print \text{a} do the same for the previous argument; if argument \text{eqn} print \text{p} otherwise print \text{ie} do the same for the previous argument, respectively.

- \[ \text{R} \]
⇒ Newline. Print \text{n} newlines.

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

- \[ \text{Conditional Newline} \]
  \[ \text{Print-Newline} \]
  \[ \text{newline} \]
⇒ Conditional Newline. Print a newline like print-newline with argument \text{line}, \text{fill}, \text{spaces}, or \text{mandatory}, respectively.

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

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

- \[ \text{min-col} \]
  \[ \text{col-min} \]
  \[ \text{min-pad} \]
  \[ \text{pad-char} \]
⇒ \text{nl-text} \text{space} \text{width}; \text{text} \text{text} text

16
Print a conditional newline if stream is a pretty printing stream. Return NIL.

*print-array*  ➢ If T, print arrays ,readably.

*print-base*  ➢ Radix for printing rationals, from 2 to 36.

*print-case*  ➢ Print symbol names all uppercase (upcase), all lowercase (downcase), capitalized (capitalize).

*print-circle*  ➢ If T, avoid indefinite recursion while printing circular structure.

*print-escape*  ➢ If NIL, do not print escape characters and package prefixes.

*print-gensym*  ➢ If T, print #: before uninterned symbols.

*print-length*  ➢ Return highest priority and if there was a matching type specifier in table.

*print-level*  ➢ Right margin width in ems while pretty-printing.

*print-miser-width*  ➢ If integer, restrict printing of objects to that number of elements per line/to that depth/to that number of lines.

*print-pretty*  ➢ If T, print prettily.

*print-radix*  ➢ If T, print rationals with a radix indicator.

*print-readably*  ➢ If T, print ,readably or signal error print-not-readable.

*print-right-margin*  ➢ Right margin width in ems while pretty-printing.

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

 pprint-dispatch foo (table)  ➢ Return highest priority function associated with type of foo and T if there was a matching type specifier in table.

(copy-pprint-dispatch (table))  ➢ 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

(format control)  ➢ Return function of stream and arg* applying format to stream, control, and arg* returning NIL or any excess args.

(format (FILL 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 ,format which is then applied to out-string and arg*. Output to out-string, out-stream or, if first argument is T, to ,standard-output*. Return NIL.

0.93 Functions

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

```
(defun foo (ord-*λ*)
  (declare (local-ord-*λ*)
    \(\lambda\) form)
  \(\lambda\) form)
```

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

```
(setf foo (new-value ord-*λ*)
  \(\lambda\) form)
```

Evaluate forms with locally defined functions foo. Globally defined functions of the same name are shadowed. Each foo is also the name of an implicit ,block around its corresponding local-ord-*λ*-. Only for ,labels, functions foo are visible inside local-forms. Return values of forms.

```
(defun \(\lambda\) form
  \(\lambda\) form)
```

Return lexically innermost function named foo or a lexical closure of the ,lambda expression.

```
(setf function arg*)
```

Values of function called with args and the list elements of args. ,setfable if function is one of ,aref, ,bit, and ,bnot.

```
(funcall function arg*)
```

Values of function called with args.

```
(multiple-value-call function form*\)
```

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

```
(values-list list)
```

Return elements of list.

```
(values foo*)\)
```

Return as multiple values the primary values of the foos. ,settable.

```
(multiple-value-list form)
```

List of the values of form.

```
(mnth-value n form)
```

Zero-indexed nth return value of form.

```
(complement function)
```

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

```
(constantly foo)
```

Function of any number of arguments returning foo.

```
(\(\lambda\) form)
```

Return foo.
(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.

(defdefinition foo (setf foo))
  ➔ Definition of global function foo. setfable.

(fmakunbound foo)
  ➔ Remove global function or macro definition foo.

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

.multiple-values-limit
  ➔ Upper bound of the number of values a multiple value can have; ≥ 20.

9.4 Macros

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

(whole var E) [macro-λ*)] [E]

&optional (var) (init-supplied-p)] [E]

&rest (rest-var) [macro-λ*)] [E]

&body (var) [macro-λ*)] [E]

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

&allow-other-keys &aux (var) [init-supplied-p)] [E] or

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

(var) [init-supplied-p)] [E] . rest-var).

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

(definemacro (define-compiler-macro) (foo (setf foo)) (macro-λ*) (declare (dec*l) (doc) form))
  ➔ 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
  block named foo.

(define-symbol-macro (foo form))
  ➔ Define symbol macro foo which on evaluation evaluates
  expanded form.

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

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

(defun (defsetf function (updater doc)
  (setf-λ*) (s-varl) (declare decl*l) (doc) form))
  ➔ where defsetf lambda list (setf-λ*) has the form (var*

\write-char char [stream standard-output])
  ➔ Output char to stream.

\write-string [write-line] string [stream standard-output] [:start start] :
  ➔ Write string to stream without/with a trailing newline.

\write-byte byte stream)
  ➔ Write byte to binary stream.

\write-sequence sequence stream [:start start] [:end end]
  ➔ Write elements of sequence to binary or character stream.

\write-to-string foo
  ➔ Print foo to stream and return foo, or print foo into
  string, respectively, after dynamically setting printer vari-
  ables corresponding to keyword parameters (\print-bar
  becoming :bar). (\stream keyword with \write only.)

\print-fill stream foo [parenthesis] [noop]]

\print-tabular stream foo [parenthesis] [noop] [nu])

\print-linear stream foo [parenthesis] [noop]]
  ➔ Print foo to stream. If foo is a list, print as many elements
  per line as possible; do the same in a table with a column
  width of n ens; or print either all elements on one line or
  each on its own line, respectively. Return NIL. Usable with
  /format directive ~/f.

\print-logical-block (stream list [:prefix string]
  ➔ 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 \write. Return NIL.

\print-pop
  ➔ Take next element off list. If there is no remaining
tail of list, or *\print-lengths* or *\print-circles* indicate
  printing should end, send element together with an ap-
  propriate indicator to stream.

\print-tab \line \line-relative \section \section-relative
  ➔ Move cursor forward to column number c + ki, k ≥ 0
  being as small as possible.

\print-indent [block] [current] \n [stream standard-output])
  ➔ Specify indentation for innermost logical block rel-
  ative to leftmost position/to current position. Return
  NIL.

\print-exit-if-list-exhausted
  ➔ If list is empty, terminate logical block. Return NIL
  otherwise.
Then

\[ f \text{ nil} \]

arg-vars

test

]\)

(*complex a b), the complex number \( a + bi \).

(function foo); the function named foo.

\( [\text{n}] \) (foo*)

Vector of some (or \( n \)) foo filled with last \( \text{foo} \) if necessary.

\( [\text{n}] \) (b*)

Bit vector of some (or \( n \)) bs filled with last \( b \) if necessary.

\( \text{S}(\text{type} \{\text{slot value}\}) \)

Structure of type.

\( \text{P}(\text{string}) \)

A pathname.

\( \text{f}(\text{oo}) \)

Uninterned symbol foo.

\( \text{f}(\text{orm}) \)

Read-time value of form.

**read-eval-print**

If \( \text{NIL} \), a reader-error is signalled at \( \#. \).

\( \text{integer}= \text{foo} \)

Give \( \text{foo} \) the label integer.

\( \text{integer#} \)

Object labelled integer.

\( \#< \)

Have the reader signal reader-error.

+feature when-feature

-when-feature unless-feature

Means when-feature if feature is \( \text{T} \); means unless-feature if feature is \( \text{NIL} \). Feature is a symbol from **features**, or (**and**|**or**|feature*), or (not feature).

**features**

List of symbols denoting implementation-dependent features.

\( [\ast ; \c c] \)

\( [\ast ; \c c] \)

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

13.4 Printer

\{prin1\}

\{print\}

\{printf\}

\{princ\}

\{print1-to-string foo\}

\{princ-to-string foo\}

\{print-object object stream\}

\{print-unreadable-object (foo stream \{\text{:type} \text{whole} \}) \text{form}^*\}

\{terpri stream \{standard-output\}\}

\{fresh-line stream \{standard-output\}\}

\( \text{print} \) to stream, \( \text{readably} \), \( \text{readably} \) between a newline and a space, \( \text{readably} \) after a newline, or human-readable without any recognizable characters, respectively. \( \text{print1} \), \( \text{print} \) and \( \text{princ} \) return \( \text{foo} \).

Print \( \text{foo} \) to stream \( \text{readably} \) or human-readable, respectively.

Print \( \text{object} \) to stream. Called by the Lisp printer.

Enclosed in \#< and \#, print \( \text{foo} \) by means of \text{forms} to stream. Return \( \text{NIL} \).

Output a newline to stream. Return \( \text{NIL} \).

Output a newline to stream and return \( \text{T} \) unless stream is already at the start of a line.

9.5 Control Flow

\{if test then \text{else}\}

\( \text{Return values of then if test returns} \text{T}; \text{return values of else otherwise.} \)

\{ semp (test then)^*\}

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

\{when \text{unless} \text{test foo}\}^*

\( \text{Evaluate foos and return their values if test returns} \text{T} \text{or} \text{NIL}, \text{respectively. Return} \text{NIL} \text{otherwise.} \)
(mcase test (key) \( \text{foo}\) value \( \text{bar}\) \( \text{T}\))

\[ \text{\(v\)} \]

\( \text{Return the values of the first foo one of whose keys is eql test. Return values of bar if there is no matching key.} \]

(mcase test (key) \( \text{foo}\) value \( \text{bar}\) \( \text{T}\))

\[ \text{\(v\)} \]

\( \text{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)

\[ \text{\(v\)} \]

\( \text{Evaluate forms from left to right. Immediately return NIL if one form’s value is NIL. Return values of last form otherwise.} \]

(or form)

\[ \text{\(v\)} \]

\( \text{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)

\[ \text{\(v\)} \]

\( \text{Evaluate forms sequentially. Return values of last form.} \]

(multiple-value-prog1 form-r form)

\[ \text{\(v\)} \]

\( \text{Evaluate forms in order. Return values/primary value, respectively, of form-r.} \]

(multiple-value-prog2 form-r form)

\[ \text{\(v\)} \]

\( \text{Evaluate forms in order. Return values/primary value, respectively, of form-r.} \]

(unwind-protect protected cleanup)

\[ \text{\(v\)} \]

\( \text{Evaluate protected and then, no matter how control leaves protected, cleanups. Return values of protected.} \]

(block name form)

\[ \text{\(v\)} \]

\( \text{Evaluate forms in a lexical environment, and return their values unless interrupted by \text{return-from}.} \]

(return-from foo result)

\[ \text{\(v\)} \]

\( \text{Have nearest enclosing block named foo/named NIL, respectively, return with values of result.} \]

(tagbody tag form)

\[ \text{\(v\)} \]

\( \text{Evaluate forms in a lexical environment. tags (symbols or integers) have lexical scope and dynamic extent, and are targets for go. Return NIL.} \]

(go tag)

\[ \text{\(v\)} \]

\( \text{Within the innermost possible enclosing tagbody, jump to a tag, eq tag.} \]

(catch tag form)

\[ \text{\(v\)} \]

\( \text{Evaluate forms and return their values unless interrupted by \text{throw}.} \]

(throw tag form)

\[ \text{\(v\)} \]

\( \text{Have the nearest dynamically enclosing catch with a tag eq tag return with the values of form.} \]

(sleep n)

\[ \text{\(v\)} \]

\( \text{Wait n seconds; return NIL.} \]

(/read-sequence sequence stream [start start] [end end])

\[ \text{\(v\)} \]

\( \text{Replace elements of sequence between start and end with elements from binary or character stream. Return index of sequence’s first unmodified element.} \]

(/readtable-case readable immediate)

\[ \text{\(v\)} \]

\( \text{Case sensitivity attribute (one of \text{upcase, downcase, preserve, invert}) of readable. settable.} \]

(/copy-readable from-readable readable to-readable)

\[ \text{\(v\)} \]

\( \text{Return copy of from-readable.} \]

(/set-syntax-from-char to-char from-char)

\[ \text{\(v\)} \]

\( \text{Copy syntax of from-char to to-readable. Return T.} \]

(/reader*)

\[ \text{\(v\)} \]

\( \text{Current readable.} \]

(/read-base radix)

\[ \text{\(v\)} \]

\( \text{Radix for reading integers and ratios.} \]

(/read-float-format simple float)

\[ \text{\(v\)} \]

\( \text{Floating point format to use when not indicated in the number read.} \]

(/read-suppress sign)

\[ \text{\(v\)} \]

\( \text{If T, reader is syntactically more tolerant.} \]

(/set-macro-character char function [non-term get|set])

\[ \text{\(v\)} \]

\( \text{Make char a macro character associated with function of stream and char. Return T.} \]

(/get-macro-character char)

\[ \text{\(v\)} \]

\( \text{Reader macro function associated with char, and T if char is a non-terminating macro character.} \]

(/make-dispatch-macro-character char [non-term get|set])

\[ \text{\(v\)} \]

\( \text{Make char a dispatching macro character. Return T.} \]

(/set-dispatch-macro-character char sub-char function)

\[ \text{\(v\)} \]

\( \text{Make function of stream, n, sub-char a dispatch function of char followed by n, followed by sub-char. Return T.} \]

(/get-dispatch-macro-character char sub-char)

\[ \text{\(v\)} \]

\( \text{Dispatch function associated with char followed by sub-char.} \]

13.3 Character Syntax

\[ \text{\(v\)} \]

\( \text{Comments. There are stylistic conventions:} \]

\(
\begin{array}{ll}
\text{;;; title} & \text{Short title for a block of code.} \\
\text{;;; intro} & \text{Description before a block of code.} \\
\text{;;; state} & \text{State of program or of following code.} \\
\text{; explanation} & \text{Regarding line on which it appears.} \\
\text{; continuation} & \text{Begin and end of a string.} \\
\end{array}
\)

\[ \text{\(v\)} \]

\( \text{List of foos with the terminating cdr bar.} \]

\[ \text{\(v\)} \]

\( \text{foo} \)

\[ \text{\(v\)} \]

\( \text{\text{quote} foo}; \text{foo unevaluated.} \]

\[ \text{\(v\)} \]

\( \text{\text{quote} foo and bing}; \text{evaluate bar and splice the lists baz and quux into their elements. When nested, outermost conses inside the innermost backquote expression belong to this backquote.} \]

\[ \text{\(v\)} \]

\( \text{\text{character} \text{"c"}, the character c.} \]

\[ \text{\(v\)} \]

\( \text{Integer of radix 2, 8, 10, 16, or r; } 2 \leq r \leq 36. \]

\[ \text{\(v\)} \]

\( \text{Integer of radix 2, 8, 10, 16, or r; } 2 \leq r \leq 36. \]
13 Input/Output

13.1 Predicates

(streamp foo)  \( \Rightarrow \) T if foo is of indicated type.

(pathnamematchp foo)  \( \Rightarrow \) T if foo is of indicated type.

(readtablep foo)  \( \Rightarrow \) T if foo is of indicated type.

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

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

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

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

(pathnamematchp path wildcard)  \( \Rightarrow \) T if path matches wildcard.

(wild-pathname-p path [ :host | :device | :directory | name | type | version | nil])  \( \Rightarrow \) Return T if indicated component in path is wildcard. (NIL indicates any component.)

13.2 Reader

(y-or-n-p)  \( \{ \) control arg\( \}) \( \Rightarrow \) Ask user a question and return T or NIL depending on their answer. See page 36, format, for control and args.

(yes-or-no-p)  \( \{ \) control arg\( \}) \( \Rightarrow \) Ask user a question and return T or NIL depending on their answer. See page 36, format, for control and args.

(with-standard-io-syntax form*)  \( \Rightarrow \) Evaluate forms with standard behaviour of reader and printer. Return values of forms.

(read-preserving-whitespace)  \( \{ \) stream \( \Rightarrow \) [eof-error | [eof-value [recursive nil]]]] \( \Rightarrow \) Read printed representation of object.

(read-from-string string [ [eof-error | [eof-value [start start]]] [end end] [preserve-whitespace bool]]])  \( \Rightarrow \) Return object read from string and zero-indexed position of next character.

(read-delimited-list char [stream [standard-input] [recursive nil]]])  \( \Rightarrow \) Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.

(read-char [stream [standard-input] [recursive nil]]])  \( \Rightarrow \) Return next character from stream.

(read-char-no-hang [stream [standard-input] [eof-error [eof-value [recursive nil]]] nil])  \( \Rightarrow \) Next character from stream or NIL if none is available.

(peek-char [mode stream [standard-input] [eof-error [eof-value [recursive nil]]]])  \( \Rightarrow \) Next, or if mode is T, next non-whitespace character, or if mode is a character, next instance of it, from stream without removing it there.

(unread-char character [stream [standard-input]]])  \( \Rightarrow \) Put last read character back into stream; return NIL.

(read-byte stream [eof-error [eof-value nil]])  \( \Rightarrow \) Read next byte from binary stream.

(read-line [stream [standard-input] [recursive nil]])  \( \Rightarrow \) Return a line of text from stream and T if line has been ended by end of file.

9.6 Iteration

\{do | loop | while | until | for\} \{variant \[step step]\} \{ (stop result) \} \{ (declare decl) \} \{tag \} \{form \}

\( \Rightarrow \) 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 \( stop \) is T. Return values of result*. Implicitly, the whole form is a \( \block \) named NIL.

(dotimes \( \{ \) var \( \} \{ result \} \{ (declare decl) \} \{ (tag \} \{form \} \}

\( \Rightarrow \) Evaluate \( \tagbody \) like body with var successively bound to integers from 0 to i - 1. Upon evaluation of result, var is i. Implicitly, the whole form is a \( \block \) named NIL.

(dolist \( \{ \) var list \( \} \{ result \} \{ (declare decl) \} \{ (tag \} \{form \} \}

\( \Rightarrow \) Evaluate \( \tagbody \) like body with var successively bound to elements of list. Upon evaluation of result, var is NIL. Implicitly, the whole form is a \( \block \) named NIL.

9.7 Loop Facility

\{loop form*\}

\( \Rightarrow \) Simple Loop. If forms do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit \( \block \) named NIL.

\{loop clause*\}

\( \Rightarrow \) Loop Facility. For Loop Facility keywords see below and Figure 1.

d named \( \text{em} \) \( \Rightarrow \) Give \( \text{loop} \)’s implicit \( \block \) a name.

\{with \[ \{ \} \] \{d-type \} \{ \} \} \text{\em} \{and \[ \{ \} \] \{d-type \} \{ \} \}

\( \Rightarrow \) Initialize (possibly trees of) local variables \( \text{var-s} \) sequentially and \( \text{var-p} \) in parallel.

\{for \[ \{ \} \] \{d-type \} \{ \} \} \text{\em} \{and \[ \{ \} \] \{d-type \} \{ \} \}

\( \Rightarrow \) Begin of iteration control clauses. Initialize and step (possibly trees of) local variables \( \text{var-s} \) sequentially and \( \text{var-p} \) in parallel. Deconstructing type specifier \( d-type \) as with

\{upfrom \} \text{\em} \{downfrom \} \text{\em} \{start \}

\( \Rightarrow \) Start stepping with start

\{upto \} \text{\em} \{downto \} \{tobelow \} \{above \} \text{\em} \{form \}

\( \Rightarrow \) Specify form as the end value for stepping.

\{in \} \text{\em} \{list \}

\( \Rightarrow \) Bind \text{var} to successive elements/tails, respectively, of \text{list}.

by \{step \} \text{\em} \{function \} \text{\em} \{on \}

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

\( = \) \text{\em} \{then \} \text{\em} \{bar \}

\( \Rightarrow \) Bind \text{var} initially to \text{foo} and later to \text{bar}.

across \text{vector}

\( \Rightarrow \) Bind \text{var} to successive elements of \text{vector}.

being \{the \} \text{each}

\( \Rightarrow \) Iterate over a hash table or a package.

\{hash-key \} \text{\em} \{hash-keys \} \{off \} \text{\em} \{in \} \text{\em} \{hash-table \} \{using \} \{hash-value \} \text{\em} \{value \}

\( \Rightarrow \) Bind \text{var} successively to the keys of \text{hash-table}; bind \text{value} to corresponding values.
12 Types and Classes

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

(type foo [environment])  » T if 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.

(typep foo [type-a form])
  » Return values of the first a-form" whose type is foo of. Return values of b-forms if no type matches.

(metypecase foo [type b-form]…) [otherwise b-form]
  » Return values of the first form" whose type is foo of. Signal non-correctable/correctable type-error if no type matches.

(type-of foo)  » Type of foo.

(check-type place type [string])
  » Signal correctable type-error if place is not of type. Return NIL.

(stream-element-type stream)  » Type of stream objects.

(array-element-type array)  » Element type array can hold.

(upgraded-array-element-type type [environment])
  » Element type of most specialized array capable of holding elements of type.

(defun foo (macro-λ*) (declare (doc) * [doc form]…)
  » 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 18 but with default value of * instead of NIL. forms are enclosed in an implicit block named foo.

(eq 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.

(type* list)  » Type specifier for intersection of types.

(type list)  » 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.
10 CLOS

10.1 Classes

(slot-exists-p foo bar)  \(\triangleright\) T if foo has a slot bar.

(slot-boundp instance slot)  \(\triangleright\) T if slot in instance is bound.

(defclass foo (superclass "standard-object"") (slot
  {reader reader’}
  {writer writer’}
  {accessor accessor’}
  {allocation [instance] [class] [instance]}
  {documentation slot-doc}
  {default-initargs (name value’)}
  {documentation class-doc}
  {metaclass name "standard-class"})  \(\triangleright\) Define or modify class foo as a subclass of superclasses. Transform existing instances, if any, by make-instances-obsolete. In a new instance i of foo, a slot’s value defaults to form unless set via :initarg-name; it is readable via (reader i) or (accessor i), and writable via (writer value i) or (setf (accessor i) value). slots with :allocation :class are shared by all instances of class foo.

(find-class symbol (environment))  \(\triangleright\) Return class named symbol. setfable.

(make-instance class (initarg value’) other-keyarg’)  \(\triangleright\) Make new instance of class.

(reinitialize-instance instance (initarg value’) other-keyarg’)  \(\triangleright\) Change local slots of instance according to initargs by means of shared-initialize.

(slot-value foo slot)  \(\triangleright\) Return value of slot in foo. setfable.

(slot-makunbound instance slot)  \(\triangleright\) Make slot in instance unbound.

(with-slots (var slot’) \{instance (declare decr’)
  \} \(\triangleright\) Return values of forms after evaluating them in a lexical environment with slots of instance visible as setfable slots or vars/with/arrays of instance visible as setfable vars.

(class-name class)  \(\triangleright\) Get/set name of class.

(class-of foo)  \(\triangleright\) Class foo is a direct instance of.

(change-class instance new-class (initarg value’) other-keyarg’)  \(\triangleright\) 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.

(make-instances-obsolete)  \(\triangleright\) Update all existing instances of class using update-instance-for-redefined-class.

(initialize-instance instance)
  \(\triangleright\) Set slots on behalf of make-instance of change-class by means of shared-initialize.

(make-instance纲 class)  \(\triangleright\) Make a direct instance of.

(update-instance-for-different-class previous current)
  \(\triangleright\) Set slots on behalf of make-instance of change-class by means of shared-initialize.

(update-instance-for-redefined-class)
  \(\triangleright\) Update all existing instances of class using make-instance.

### Common Lisp Quick Reference

(\(\text{\texttt{(m-restart-bind \{\text{\textit{restart} \text{\textit{restart-function}}\}} \text{\textit{form}}\))}\)
\(\triangleright\) 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 any-function.

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

(find-restart (compute-restarts name) \{condition\})
\(\triangleright\) 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.

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

\(\text{\texttt{(m-restart-bind \{\text{\textit{restart} \text{\textit{restart-function}}\}} \text{\textit{form}}\))}\)
\(\triangleright\) Evaluate forms with restarts dynamically associated with condition. Return values of forms.

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

(cell-error-name condition)
\(\triangleright\) Name of cell which caused condition.

(unbound-slot-instance condition)
\(\triangleright\) Instance with unbound slot which caused condition.

(print-not-readable-object condition)
\(\triangleright\) The object not readable printable under condition.

(package-error-package condition)
\(\triangleright\) The file-error-pathname condition.

(stream-error-stream condition)
\(\triangleright\) Package, path, or stream, respectively, which caused the condition of indicated type.

(type-error-datum condition)
\(\triangleright\) Object which caused condition of type type-error, or its expected type, respectively.

(simple-condition-format-control condition)
\(\triangleright\) Return format control or list of format arguments, respectively, of condition.

(simple-condition-format-arguments condition)
\(\triangleright\) Condition type debugger is to be invoked on.
(!make-condition condition-type ![binary-name value]!)  
  \(\Rightarrow\) Return new instance of condition-type.

(!signal)  
  condition ![binary-name value]!  
  \(\Rightarrow\) 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 36), simple-condition, simple-warning, or simple-error, respectively. From signal and warn, return NIL.

(!error)  
  continue-control  
  condition continue-arg*  
  condition-type ![binary-name value]!  
  \(\Rightarrow\) Unless handled, signal as correctable error condition or a new instance of condition-type or, with format control and args (see page 36), simple-condition, simple-warning, or simple-error, respectively. In the debugger, use format arguments continue-control and continue-args to tag the continue option. Return NIL.

(!ignore-errors form)  
  \(\Rightarrow\) Return values of forms or, in case of errors, NIL and the condition.

(!invoke-debugger condition)  
  \(\Rightarrow\) Invoke debugger with condition.

(!assert test ![place*])  
  condition continue-arg*  
  condition-type ![binary-name value]!  
  \(\Rightarrow\) 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 36), 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])  
  ![no-error](ord-lambda) ![condition-form]!  
  \(\Rightarrow\) 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-lambda to value of foo and return values of forms or, without a no-error clause, return values of foo. See page 17 for (ord-lambda).

(!handler-bind ![condition-type handler-function]! form)  
  \(\Rightarrow\) Return values of forms after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.

(!with-simple-restart ![restart]! ![control arg]*) form  
  \(\Rightarrow\) Return values of forms unless restart is called during their evaluation. In this case, describe restart using #format control and args (see page 36) and return NIL and T.

(!restart-case form ![restart ord-lambda])  
  ![interactive-arithmetic-function]!  
  ![report-function]!  
  ![test-function]!  
  \(\Rightarrow\) 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-lambda, or by (invoke-restart-interactively restart) where a list of the respective args is supplied by #*arg-function. See page 17 for ord-lambda.

(!update-instance-for-redefined-class new-instance added-slots discarded-slots discarded-slots-property-list ![binary-name value]! ![other-kegpar]!)  
  \(\Rightarrow\) On behalf of &make-instances-obsolele and by means of #shared-initialize, set any instance to its corresponding values; set any remaining added-slots to the values of their :initialize forms. Not to be called by user.

(!allocate-instance ![binary-name value]! ![other-kegpar]!)  
  \(\Rightarrow\) Return uninitialized instance of class. Called by #make-instance.

(!slot-missing class instance slot ![slot-boundp]! ![slot-makunbound]! ![slot-value])  
  \[value]\]  
  \(\Rightarrow\) 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

(!next-method-?)  
  \(\Rightarrow\) T if enclosing method has a next method.

(!defgeneric foo ![setf foo]) ![required-var] ![optional var]!  
  \(\Rightarrow\) Define 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 defmethod. For c-type see section 10.3.

(!ensure-generic function ![setf function])  
  ![required-var] ![optional var]!  
  \(\Rightarrow\) Define or modify generic function function. 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 defmethod. For c-type see section 10.3.
Define new method for generic function `foo`. `spec-vars` specialize to either being of class or being equal to `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.

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

Return suitable method, or signal `error`.

List of methods suitable for `args`, most specific first.

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

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

Signal `error` on applicable method with invalid qualifiers, or on method combination. For `control` and `args` see `format`, page 36.

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

Return list of keyword parameters of `method` and `allow-other-keys` if other keys are allowed.

List of qualifiers of `method`.

**Method Combination Types**

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.

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

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 primary 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)` (specified as `c-type` in `mdefine-generic`). Using `c-type` as the qualifier in `mdefine-method` makes the method primary.

A call to a generic function using `c-type` will be equivalent to a call to the forms returned by `body` with `ord-lambda` bound to `c-arg` (cf. `mdefine-generic`), with `symbol` bound to the generic function, with `method-combination-lambda` bound to the arguments of the generic function, and with `groups` bound to lists of methods. An applicable method becomes a member of the leftmost group whose `predicate` or `qualifiers` match. Methods can be called via `mcall-method`. Lambda lists (ord-lambda) and (method-combination-lambda) according to ord-lambda on page 17, the latter enhanced by an optional `&whole` argument.

For standardized condition types cf. Figure 2 on page 31.

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