Contents

1 Numbers 3
1.1 Predicates . . . . 3
1.2 Numeric Functns . 3
1.3 Logic Functions . 5
1.4 Integer Functions . 6
1.5 Implementation-Dependent . . . 6
2 Characters 6
3 Strings 7
4 Conses 8
4.1 Predicates . . . . 8
4.2 Lists . . . . . 9
4.3 Association Lists . 10
4.4 Trees . . . . . 10
4.5 Sets . . . . . 11
5 Arrays 11
5.1 Predicates . . . . 11
5.2 Array Functions . 11
5.3 Vector Functions . 12
6 Sequences 12
6.1 Seq. Predicates . 12
6.2 Seq. Functions . 13
7 Hash Tables 15
8 Structures 16
9 Control Structure 16
9.1 Predicates . . . . 16
9.2 Variables . . . . 17
9.3 Functions . . . . 17
9.4 Macros . . . . . 19
10 CLOS 24
10.1 Classes . . . . . 24
10.2 Generic Functns . 26
10.3 Method Combination Types . . . 27
11 Conditions and Errors 28
12 Input/Output 30
12.1 Predicates . . . . 30
12.2 Reader . . . . . 31
12.3 Macro Chars . . 32
12.4 Printer . . . . . 33
12.5 Format . . . . . 35
12.6 Streams . . . . . 37
12.7 Files . . . . . 39
13 Types and Classes 40
14 Packages and Classes 42
15 Compiler 44
15.1 Predicates . . . . 44
15.2 Compilation . . 45
15.3 REPL & Debug . 46
15.4 Declarations . . 47
16 External Environment 47

Typographic Conventions

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

them ▷ Placeholder for actual code.
me ▷ Literal text.
[foo; bar] ▷ Either 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} ▷ Either foo, or bar, or 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 prog; see p. 20.
foo; bar; baz ▷ First, second and nth return value.
T; NIL ▷ t, or truth in general; and nil or ().
1 Numbers

1.1 Predicates

\( p \leq r \) if \( a \leq b \), \( a \leq b \), \( a > b \), \( a < b \)

\( \log a [b] \) if \( a \) or \( b \), \( a \) or \( b \), \( a \) or \( b \), \( a \) or \( b \)

\( \sqrt n \) in complex or natural numbers, respectively.

\( \text{gcd} a \) if \( \text{least common multiple} \) or \( \text{greatest common denominator} \), respectively, of \( a \). \( \text{gcd} a \) returns \( 0 \).
\( \text{sinh} \ a \) \( \rightarrow \text{sinh} \ a \), \( \text{cosh} \ a \), or \( \text{tanh} \ a \), respectively.

\( \text{atan} \ a \) \( \rightarrow \text{atan} \ a \), \( \text{acos} \ a \), or \( \text{atanh} \ a \), respectively.

\( \text{asin} \ a \) \( \rightarrow \text{asin} \ a \), \( \text{acos} \ a \), or \( \text{atan} \ a \), respectively.

\( \text{mod} \) \( n \ d \) \( \rightarrow \text{Same as } \text{floor} \text{ or } \text{truncate}, \text{respectively, but return remainder only.} \)

\( \text{random} \) \( \text{limit [state]} \) \( \rightarrow \text{Return non-negative random number less than } \text{limit}, \text{of the same type.} \)

\( \text{make-random-state} \) \( \text{[state NLL | LAM]} \) \( \rightarrow \text{Copy of random-state object } \text{state or of the current random state, or a randomly initialized fresh random state.} \)

\( \text{random-state} \) \( \rightarrow \text{Current random state.} \)

\( \text{floor} \) \( \text{num} \) \( \rightarrow \text{num} \) \( \text{with the sign of } \text{num} \text{-a.} \)

\( \text{signum} \) \( n \) \( \rightarrow \text{Number of magnitude 1 representing sign or phase of } n. \)

\( \text{numerator rational} \) \( \text{denominator rational} \) \( \rightarrow \text{Numerator or denominator, respectively, of rational's canonical form.} \)

\( \text{realpart number} \) \( \text{imagpart number} \) \( \rightarrow \text{Real part or imaginary part, respectively, of number.} \)

\( \text{complex real [imag] \) \( \rightarrow \text{Make a complex number.} \)

\( \text{phase number} \) \( \rightarrow \text{Angle of number's polar representation.} \)

\( \text{abs} \) \( n \) \( \rightarrow \text{Return } |n|. \)

\( \text{rational real} \) \( \text{rationalize real} \) \( \rightarrow \text{Convert real to rational. Assume complete/limited accuracy for real.} \)

\( \text{float real [prototype imag-real]} \) \( \rightarrow \text{Convert real into float with type of prototype.} \)
1.3 Logic Functions

Negative integers are used in two’s complement representation.

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

boole-1 ➞ int-a
boole-2 ➞ int-b
boole-c1 ➞ ~int-a
boole-c2 ➞ int-a ∧ int-b
boole-set ➞ All bits set.
boole-dr ➞ All bits zero.
boole-eqv ➞ int-a ≡ int-b.
boole-and ➞ int-a ∧ int-b
boole-andc1 ➞ ~int-a ∧ ~int-b
boole-andc2 ➞ ~int-a ∧ ~int-b
boole-nand ➞ ~(int-a ∧ int-b)
boole-orc1 ➞ int-a ∨ int-b
boole-orc2 ➞ ~int-a ∨ ~int-b
boole-xor ➞ ~int-a ⊻ ~int-b
boole-nor ➞ ~(int-a ∨ int-b).

(lognot integer) ➞ ~integer.

(logeqv integer*) ➞ Return value of exclusive-nored or anded integers, respectively. Without any integer, return \(-1\).

(logandc1 int-a int-b) ➞ ~int-a ∧ ~int-b
(logandc2 int-a int-b) ➞ ~int-a ∧ int-b

(lognand int-a int-b) ➞ ~int-a ∧ ~int-b

(logxor integer*) ➞ Return value of exclusive-ored or ored integers, respectively. Without any integer, return \(0\).

(logorc1 int-a int-b) ➞ int-a ∨ ~int-b
(logorc2 int-a int-b) ➞ int-a ∨ ~int-b

(lognort int-a int-b) ➞ ~int-a ∨ ~int-b.

(lobitp i integer) ➞ T if zero-indexed ith bit of integer is set.

(logtest int-a int-b) ➞ Return T if there is any bit set in int-a which is set in int-b as well.

(logcount int) ➞ Number of 1 bits in int ≥ 0, number of 0 bits in int < 0.

(ash integer count) ➞ Return copy of integer arithmetically shifted left by count adding zeros at the right, or, for count < 0, shifted right discarding zeros.

(mask-field byte-spec integer) ➞ Return copy of integer with all bits unset but those denoted by byte-spec. settable.
1.4 Integer Functions

(integer-length integer)  
▷ Number of bits necessary to represent integer.

(byte-test byte-spec integer)  
▷ Return 1 if any bit specified by byte-spec in integer is set.

(byte-test integer)  
▷ Extract byte denoted by byte-spec from integer. settable.

(integer-decode-float n)  
▷ Return most-negative, least-negative-normalized, least-positive, least-positive-normalized.

(integer-decode-float n)  
▷ Smallest possible number making a difference when added or subtracted, respectively.

(short-float)  
▷ Available numbers closest to 0.

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

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

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

2 Characters

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

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

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, ignorable, ignore, inline, notinline, optimize, or special. See below.

(declaration foo*)  
▷ Make foos names of declarations.

(dynamic-extent variable* (function function)* )  
▷ Declare lifetime of variables and/or functions to end when control leaves enclosing block.

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

(ignore function)  
▷ Tell compiler how to optimize. n = 0 means unimportant, n = 1 is neutral, n = 3 means important.

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

(optimize compilation-speed)  
▷ Tell compiler how to optimize. n = 0 means unimportant, n = 1 is neutral, n = 3 means important.

(16 External Environment

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

(get-internal-run-time)  
▷ Current time, or computing time, respectively, in clock ticks.

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

(encode-universal-time see min hour date month year [zone])  
▷ Seconds from 1900-01-01, 00:00.

(get-universal-time)  
▷ Seconds from 1900-01-01, 00:00.

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

(room [[default]])  
▷ Print information about internal storage management.

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

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

(hasp-implementation software machine)  
▷ Name or version of implementation, operating system, or hardware, respectively.

(machine-instance)  
▷ Computer name.
15.3 REPL and Debugging

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

(stringp foo) ▷ T if foo is of indicated type.

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

(lower-case-p character) ▷ Return T if character is lower-case.

(upper-case-p character) ▷ Return T if character is upper-case.

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

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

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

(char/= character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

(char> character) ▷ Return T if character is upper-case, lower-case, or able to be in another case, respectively.

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

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

(char? character) ▷ Return T if all characters, or none, respectively, are equal.

(char-equal character) ▷ Return T if all characters, or none, respectively, are equal ignoring case.

(char-not-equal character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-greaterp character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-lessp character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

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

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


(char-code character) ▷ Character with code.

(code-char code) ▷ Character with code.

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

(character c) ▷ Return #\c.

3 Strings

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

(stringp foo) ▷ T if foo is of indicated type.

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

(lower-case-p character) ▷ Return T if character is lower-case.

(upper-case-p character) ▷ Return T if character is upper-case.

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

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

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

(char/= character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

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

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

(char-equal character) ▷ Return T if all characters, or none, respectively, are equal ignoring case.

(char-not-equal character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-greaterp character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-lessp character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

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

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

(stringp foo) ▷ T if foo is of indicated type.

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

(lower-case-p character) ▷ Return T if character is lower-case.

(upper-case-p character) ▷ Return T if character is upper-case.

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

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

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

(char/= character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

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

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

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

(char-equal character) ▷ Return T if all characters, or none, respectively, are equal ignoring case.

(char-not-equal character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-greaterp character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-lessp character) ▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

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

(char-downcase character) ▷ Return corresponding uppercase/lowercase character, respectively.
4 Conses

4.1 Predicates

(consp foo)    ▷ Return T if foo is of indicated type.
(make-list foo) ▷ Return an empty list.
(endp list)     ▷ Return T if list/nil is NIL.
(null foo)      ▷ Return T if foo is not a cons.
(taillp foo list) ▷ Return T if foo is a tail of list.

(member foo list) {test function} {test-not function} {key function}
▷ Return tail of list starting with its first element matching foo. Return NIL if there is no such element.

15.2 Compilation

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

(compile-file {output-file out-path} {verbatim boolean} {compile-verbose boolean} {print boolean} {example print}
{external-format file-format format})
▷ Write compiled contents of file to out-path. Return true if output path or NIL, T in case of warnings or errors, T in case of warnings or errors excluding style warnings.

(compile-file-pathname file {output-file path} {other-keyargs})
▷ Pathname compile-file writes to if invoked with the same arguments.

(load path) {file-format file-format format} {if-does-not-exist boolean}
▷ Load source file or compiled file into Lisp environment. Return T if successful.

(input file) {pathname} {initial-read boolean} {true name boolean}
▷ Input file used by compile-file/by load.

(err "quote" foo)
▷ Return unevaluated foo.

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

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

(macro-function symbol {environment!})
▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. setfable.
### Symbols

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

- `(make-symbol name)`
  - Make fresh, uninterned symbol `name`.
- `(gensym)`
  - Return fresh, uninterned symbol `gensym`, with `n` from `gensym-counters`. Increment `gensym-counters`.
- `(gentemp [prefix] [package])`
  - Intern fresh symbol in `package`. Deprecated.
- `(copy-symbol symbol [props])`
  - Return uninterned copy of `symbol`. If `props` is `T`, give copy the same value, function and property list.
- `(symbol-name symbol)`
- `(symbol-package symbol)`
- `(symbol-plist symbol)`
- `(symbol-value symbol)`
- `(symbol-function symbol)`
  - Name, package, property list, value, or function, respectively, of `symbol`. setfable.

- `(documentation [setf documentation] new-doc foo)`
  - Set/get documentation string of `foo` of given type.

- `(truth)`
  - Truth; the supertype of every type including `t`; the superclass of every class except `t`, `*terminal-io*`.
- `(false)`
  - Falsy; the empty list; the empty type, subtype of every type; `*standard-input*`, `*standard-output*`; the global environment.

### 14.4 Standard Packages

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

### 15 Compiler

#### 15.1 Predicates

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

- `(member-if [member-if-not] test list [key function])`
  - Return tail of `list` starting with its first element satisfying `test`. Return `NIL` if there is no such element.

- `(subseq list-a list-b)`
  - Return `T` if `list-a` is a subset of `list-b`.

### 4.2 Lists

- `(cons foo bar)`
  - Return new cons `(foo , bar)`.
- `(list foo+)`
  - Return list of `foo`s.
- `(list+ foo+)`
  - Return `NIL` if `foo+` has only one `foo` given.
- `(make-list num [initial-element form])`
  - New list with `num` elements set to `form`.
- `(list-length list)`
  - Length of `list`; `NIL` for circular `list`.
- `(list-0 list)`
  - `car` of `list` or `NIL` if `list` is `NIL`. setfable.
- `(cdar list)`
  - `cdr` of `list` or `NIL` if `list` is `NIL`. setfable.
- `(nthcdr n list)`
  - Return `NIL` after calling `cdar` `n` times.
- `(list[0] [first][second][third][fourth][fifth][sixth]...)[nth][enth] list)`
  - Return `nth` element of `list` if any, or `NIL` otherwise. setfable.
- `(nth n list)`
  - Return zero-indexed `nth` element of `list`. setfable.
- `(cadr list)`
  - `cdr` of `list` or `NIL` for circular `list`.
- `(last list (numy))`
  - Return list of last `num` conses of `list`.
- `(butlast list [numy])`
  - Return list excluding last `num` conses.
- `(placa con object)`
  - Replace `car`, or `cdr`, respectively, of `cons` with `object`.
- `(deq (foo))`
  - If `foo` is a tail of `list`, return preceding part of `list`. Otherwise return `foo`.
- `(adjoint foo list)`
  - Return list if `foo` is already member of `list`. If not, return `(cadr list)`.
- `(pop (place) place)`
  - Set place to `(cadr place)`, return `(car place)`.
- `(push foo place)`
  - Set place to `(cons foo place)`.
- `(pushnew foo place)`
  - Set place to `(adjoint foo place)`.
- `(append [list*] foo)`
  - Return concatenated `list`. `foo` can be of any type.
4.3 Association Lists

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

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

assoc foo alist
  [test test]
  [test-not test]
  [key function]
  > First cons whose car, or cdr, respectively, satisfies test.

(copy-alist alist)
  > Return copy of alist.

4.4 Trees

(tree-equal foo bar)
  [test test]
  [test-not test]
  > Return T if trees foo and bar have same shape and leaves satisfying Test.

(subst new old)
  [key function]
  > Make copy of tree with each subtree or leaf matching old replaced by new.

(subst-if-not new old)
  [test function]
  [test-not function]
  [key function]
  > Make copy of tree with each subtree or leaf satisfying test replaced by new.

(subsis association-list-tree)
  [test function]
  [test-not function]
  [key function]
  > Make copy of tree with each subtree or leaf matching a key in association-list replaced by that key’s value.

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

(fuse-package other-packages packages)
  > 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.

(package-name package)
  > Name of package.

(package-nicknames package)
  > List of nicknames of package.

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

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

(intern symbol package)
  > Intern or find, respectively, symbol in package. Second return value is one of internal, external, or inherited (or NIL if intern created a fresh symbol).

(unintern symbol package)
  > Remove symbol from package, return T on success.

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

(shadow symbols package)
  > Add symbols to shadowing list of package making equally named inherited symbols shadowed. Return T.

(package-shadowing-symbols package)
  > List of shadowing symbols of package.

(export symbols package)
  > Make symbols external to package. Return T.

(unexport symbols package)
  > Revert symbols to internal status. Return T.

(do-symbols)
  [var package result]
  > Evaluate tagbody-like body with var successively bound to every symbol from package, to every external symbol from other package, or to every symbol from all registered packages, respectively. Return values of result. Implicitly, the whole form is a block named NIL.

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

\[
\begin{align*}
\text{intersection} & \quad \{a \mid (a \in a) \land (a \in b)\} \\
\text{set-difference} & \quad \{a \mid (a \in a) \land (a \not\in b)\} \\
\text{set-exclusive-or} & \quad \{a \mid (a \in a) \land (a \not\in b)\} \\
\text{nintersection} & \quad \{a \mid (a \not\in a) \lor (a \not\in b)\} \\
\text{nunion} & \quad \{a \mid (a \not\in a) \lor (a \not\in b)\} \\
\text{nset-exclusive-or} & \quad \{a \mid (a \not\in a) \land (a \not\in b)\}
\end{align*}
\]

\[\Rightarrow \text{Return } a \cap b, a \setminus b, a \cup b, \text{ or } a \triangle b, \text{ respectively, of lists } a \text{ and } b.\]

5 Arrays

5.1 Predicates

\[
\begin{align*}
\text{arrayp} & \quad \text{If } foo \text{ is of indicated type.} \\
\text{vectorp} & \quad \text{If } foo \text{ is of indicated type.} \\
\text{simple-vector-p} & \quad \text{If } foo \text{ is of indicated type.} \\
\text{bit-vector-p} & \quad \text{If } foo \text{ is of indicated type.} \\
\text{adjustable-array-p} & \quad \text{If } array \text{ is adjustable/has a fill pointer, respectively.} \\
\text{array-has-fill-pointer-p} & \quad \text{If } array \text{ has a fill pointer.} \\
\text{array-in-bounds} & \quad \text{If } array \text{ is within bounds.}
\end{align*}
\]

5.2 Array Functions

\[
\begin{align*}
\text{make-array} & \quad \text{Create or modify } array. \\
\text{adjust-array} & \quad \text{Adjust } array. \\
\text{element-type} & \quad \text{Return element type of } array. \\
\text{fill-pointer} & \quad \text{Return fill pointer of } array. \\
\text{initial-element} & \quad \text{Return initial element of } array. \\
\text{initial-contents} & \quad \text{Return initial contents of } array. \\
\text{displaced-to} & \quad \text{Return displaced-to of } array. \\
\text{displaced-index-offset} & \quad \text{Return displaced-index-offset of } array.
\end{align*}
\]

\[\Rightarrow \text{Return fresh, or readjust, respectively, vector or array of }\]

\[\text{dimensions.}\]

\[
\begin{align*}
\text{setf array} & \quad \text{Set array to }\]
\text{subscripts.} \quad \text{setfable.} \\
\text{row-major-setf} & \quad \text{Set rth element of } array \text{ in row-major order.} \quad \text{setfable.} \\
\text{row-major-index} & \quad \text{Index of } r\text{th element of } array. \\
\text{array-dimensions} & \quad \text{List containing the lengths of } array\text{'s dimensions.} \\
\text{array-dimension} & \quad \text{Length of rth dimension of } array. \\
\text{array-total-size} & \quad \text{Number of elements in } array. \\
\text{array-rank} & \quad \text{Number of dimensions of } array. \\
\text{array-displacement} & \quad \text{Target array and offset.} \\
\text{bit-array} & \quad \text{Return element of } bit-array \text{ or of } simple-bit-array. \quad \text{setfable.}
\end{align*}
\]
(bit-not bit-array [result-bit-array NIL])
▷ Return result of bitwise negation of bit-array. If result-bit-array is T, put result in bit-array; if it is NIL, make a new array for result.

(bit-eqv bit-and bit-andc1 bit-andc2 bit-nand bit-ior bit-orc1 bit-orc2 bit-xor bit-nor)
▷ Return result of bitwise logical operations (cf. operations of boolean, p. 5) on bit-array-a and bit-array-b. If result-bit-array is T, put result in bit-array-a; if it is NIL, make a new array for result.

array-rank-limit ▷ 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.

(soref vector i) ▷ Return ith element of vector. setfable.

(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. setfable.

6 Sequences
6.1 Sequence Predicates
(\*\* every notevery \*)
▷ Return NIL or T, respectively, as soon as test on any set of corresponding elements of sequences returns NIL.

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

<table>
<thead>
<tr>
<th>T</th>
<th>number</th>
<th>real [lower-limit [upper-limit]]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rational [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>integer [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>signed-byte [size bit]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned-byte [size bit]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fixnum: : bignum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td></td>
</tr>
<tr>
<td></td>
<td>float [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>short-n-fool [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>single-fool [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double-fool [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>long-fool [lower-limit [upper-limit]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>complex [type]</td>
<td></td>
</tr>
<tr>
<td>sequence</td>
<td>list</td>
<td>cons [car-type [cdr-type]]</td>
</tr>
<tr>
<td></td>
<td>symbol [null]</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>[type] [rank] [dimension *]]</td>
<td></td>
</tr>
<tr>
<td>simple-array</td>
<td>[type] [rank] [dimension *]]</td>
<td></td>
</tr>
<tr>
<td>vector</td>
<td>[type] [size]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simple-bit-vector [size]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>string [size]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simple-string [size]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>base-string [size]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simple-base-string [size]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>simple-vector [size]</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>[type] [rank] [dimension *]]</td>
<td></td>
</tr>
<tr>
<td>simple-array</td>
<td>[type] [rank] [dimension *]]</td>
<td></td>
</tr>
<tr>
<td>character</td>
<td>base-char</td>
<td>standard-char</td>
</tr>
<tr>
<td></td>
<td>extended-char</td>
<td></td>
</tr>
<tr>
<td>symbol</td>
<td>logical-pathname</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>[arg-types] [value-types]</td>
<td></td>
</tr>
<tr>
<td>generic-function</td>
<td>standard-generic-function</td>
<td></td>
</tr>
<tr>
<td>compiled-function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>standard-object</td>
<td>class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>standard-class: built-in-class: structure-class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>method</td>
<td></td>
</tr>
<tr>
<td></td>
<td>standard-method</td>
<td></td>
</tr>
</tbody>
</table>

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

As a type argument, • means no restriction.

Figure 3: Data Types.
13 Types and Classes

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

- (typep foo type)   ReturnValue if foo is of type.
- (subtypep type-a type-b [environment])   ReturnValue T if type-a is a recognizable subtype of type-b, and NIL if the relationship could not be determined.
- (the type form)   ReturnValue values of form which are declared to be of type.
- (coerce object type)   Coerce object into type.
- (typecase foo (type a-form)* [ (otherwise T) b-form])   ReturnValue values of the a-forms whose type is foo of. Return values of b-forms if no type matches.
- (typecase) foo (type form)*   ReturnValue values of the forms whose type is foo of. Signal correctable/non-correctable error, respectively if no type matches.
- (type-of foo)   Type of foo.

6.2 Sequence Functions

- (make-sequence sequence-type size [initial-element foo])   Make sequence of sequence-type with size elements.
- (concatenate type sequence*)   Return concatenated sequence of type.
- (merge type sequence-a sequence-b test [key function])   Return interleaved sequence of type. Merged sequence will be sorted if both sequence-a and sequence-b are sorted.
- (fill sequence foo [start start] [end end])   Return sequence after setting elements between start and end to foo.
- (length sequence)   ReturnValue length of sequence (being value of fill pointer if applicable).
- (count foo sequence)   ReturnValue number of foos in sequence which satisfy tests.
- (count-if) test sequence [from-end boolean] [test function] [key function]   ReturnValue number of elements in sequence which satisfy test.
- (elt sequence index)   ReturnValue element of sequence pointed to by zero-indexed index. setfable.
- (subseq sequence start [end])   Return subsequence of sequence between start and end. setfable.
- (sort sequence test [key function])   Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.
- (reverse sequence)   Return sequence in reverse order.
▷ Return first element in sequence which satisfies test, or its position relative to the begin of sequence, respectively.

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

▷ Make copy of sequence without elements matching foo.

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

▷ Make copy of sequence without duplicates.

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

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

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

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

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

▷ Bidirectional streams for debugging and user interaction.

### 12.7 Files

- `host host`
- `device dev`
- `directory dir`
- `name name`
- `version ver`
- `defaults path`
- `case :local :common scheme`

▷ Construct pathname.

▷ Return pathname after filling in missing parts from defaults.

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

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

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

▷ Return pathname component.

▷ Logical name of path.

▷ Translate path-a from wildcard path-b into wildcard path-c. Return new path.

▷ host's list of translations. setfable.

▷ Load host's translations. Return NIL if already loaded, return T if successful.
(make-concatenated-stream input-stream*)
(make-broadcast-stream output-stream*)
(make-two-way-stream input-stream-part output-stream-part)
(make-echo-stream from-input-stream-to-output-stream)
(make-synonym-stream variable-bound-to-stream)
  ▷ Return stream of indicated type.

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

(make-string-output-stream [element-type type-decl] val)
  ▷ Return a string-stream accepting type-decl (available via get-output-stream-string).

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

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

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

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

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

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

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

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

(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] [end end]) (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-input-to-string (foo string [element-type type-decl]) (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.

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

(reduce function sequence (from-end bool) [start start] [end end] [key function])
  ▷ Starting with the first two elements of sequence, apply function successively to its last return value together with the next element of sequence. Return last value of function.

(copy-seq sequence)
  ▷ Return copy of sequence with shared elements.

7 Hash Tables

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

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

(make-hash-table [test (eq (eq) equal equalp equalp) (...)]
  ▷ Make a hash-table.

(gethash key hash-table [default nil])
  ▷ Return object with key if any or default otherwise; and T otherwise.

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

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

(clear hash-table)
  ▷ Empty hash-table.

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

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

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

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

(sxhash foo)
  ▷ Hash code unique for any argument foo.
8 Structures

(defstruct (foo) (foo)
  (con-name (slot-prefix foo))
  (constructor (make foo (ord-λ)))
  (copier (copy foo))
  (include struct (slot inst (type type)))
  (type list (vector inst (vector size)))
  (named (initial-offset nil))
  (print-object (o-printer))
  (print-function (f-printer))
  (predicate (p-named nil)))

(doc {foo} (slot inst (type type))

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

(defun 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.

(equalityP 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 eql elements; or are types of the same type with equals elements; or are hash-tables of the same size with the same :test function, the same keys in terms of :test function, and equals elements.

(not foo) ⇒ t if foo is NIL, NIL otherwise.

(boundp symbol) ⇒ t if symbol is a special variable.

(constantP foo "environment") ⇒ t if foo is a constant form.

12.6 Streams

(defun open-path (path)

 ⇒ Open file-stream to path.

Common Lisp Quick Reference
- [radius | [width | [pad-char | [comma-char |]
  | comma-interval | ] | ; R]
  | (One or more prefix arguments.) Print argument as number; with ;, group digits comma-interval each; with 0, always prepend a sign.

- [R | R | R | R]
  | Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

- [width | [pad-char | [comma-char |]
  | comma-interval | ] | ; ] B/]
  | Print integer argument as number (decimal, binary, octal, or hexadecimal, respectively). With ;, group digits comma-interval each; with 0, always prepend a sign.

- [width | [dec-digits | [shift | [overflow-char |]
  | pad-char | [exp-digits |]
  | [scale-factor |]
  | [overflow-char | [pad-char | [exp-factor |]
  | E | G]
  | Print argument as floating-point number with int-digits before decimal point and exp-digits in the signed exponent. With -G, choose either -E or -F. With 0, always prepend a sign.

- [c | C | C | C | C | C]
  | Print, spell out, print in #! syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

- [dec-digits | [int-digits | [width | [pad-char |]
  | comma-interval | ] | ; ] B/S]
  | Print argument as fixed-format floating-point number. With ;, put sign before any padding; with 0, always prepend a sign.

- [text | (text | [text | text | text |]
  | ;]
  | Convert to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.

- [P | P | P | P | P]
  | If argument eql 1 print nothing, otherwise print 0; do the same for the previous argument; if argument eql 1 print y, otherwise print 1; do the same for the previous argument, respectively.

- [n | n]
  | Print n newlines.

- [n &]
  | Print n – 1 newlines if output stream is at the beginning of a line, or n newlines otherwise.

- [— — — — — —]
  | Print like pprint-newline with argument :linear, :fill, :miser, or :mandatory, respectively.

- [B | B | B | B | B]
  | Print body like pprint-logical-block using body as format control string on the elements of the list argument or, with 0, on the remaining arguments, which are extracted by pprint-pop. With ;, prefix and suffix default to ( and ). When closed by –B, spaces in body are replaced with conditional newlines.

- [B | B | B | B | B]
  | (Tilde-newline.) Ignore newline and following whitespace. With ;, ignore only newline; with 0, ignore only following whitespace.

- [text | [text | [text | text | text |]
  | ;]
  | Justify text produced by texts in a field of at least min-col columns. With ;, right justify; with 0, left justify. If this would leave less than spare characters on the current line, output nil-text first.

9.2 Variables

- [defconstant | [defparameter | [form | [doc]]
  | (doc)]
  | Assign value of form to global constant/dynamic variable.

- [defvar | [form | [doc]]
  | (doc)]
  | Unless bound already, assign value of form to dynamic variable.

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

- [setq | [symbol | form | y]
  | (y)]
  | Set symbols to primary values of forms. Return value of last form/NIL; work sequentially/in parallel, respectively.

- [set | symbol | foo]
  | Set symbol’s value cell to foo. Deprecated.

- [multiple-value-setq | vars | form | y]
  | (y)]
  | Set elements of vars to values of form. Return form’s primary value.

- [shift | place | foo | y]
  | (y)]
  | Store value of foo in rightmost place shifting values of places left, returning first place.

- [rotate | place | y]
  | (y)]
  | 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 | y]]
  | (y)]
  | First entry key from property list stored in symbol/in/place, respectively, or default if there is no key. settable.

- [get-properties | property-list | keys | y]
  | (y)]
  | 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 | y]
  | (y)]
  | Remove first entry key from property list stored in symbol/in/place, respectively. Return T if key was there, or NIL otherwise.

9.3 Functions

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

- [function | function | y]
  | (y)]
  | ✂ if foo is of type function.

- [fboundp | [foo | (setf foo)] | y]
  | (y)]
  | ✂ if foo is a global function or macro.
(deftype foo (setf foo) (or-λ*) (declare decl*) [doc] form)*

> Define a function named foo or (setf foo), or an anonymous function, respectively, which applies forms to or-λ. For defun, forms are enclosed in an implicit block foo.

(let [labels] (((foo (setf foo) [or-λ*] (declare local-decl*)*) [doc]
local-form*)*) (declare decl*)* form)

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

(function foo [lambda form]*)

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

(fapply function [setf function] arg+)

> Return values of function called on args. Last arg
must be a list. setfable if function is one of aref, bit, and sbit.

(funcall function arg*)

> Return values of function called with args.

(multiple-value-call foo form*)

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

(values-list list)

> Return elements of list.

(values foo*)

> Return as multiple values the primary values of the foo.
setfable.

(multiple-value-list form)

> Return in a list values of form.

(nth-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)

> Return function of any number of arguments returning foo.

(identity foo)

> Return function.

(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.

(definition 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.

:*print-array* → If T, print arrays readable.

:*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* →

:*print-level* →

:*print-lines* →

> If integer, restrict printing of objects to that number of
elements per level/to that depth/to that number of lines.

:*print-miser-width* → Width below which a compact pretty-printing style is used.

:*print-pretty* → If T, print pretty.

:*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 table])

> 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.

12.5 Format

(formatter control)

> Return function of stream and a &rest argument applying
format to stream, control, and the &rest argument returning
nil or any excess arguments.

(format [T|nil] out-string out-stream control arg*)

> Output string control which may contain - directives possi-
bly taking some args. Alternatively, control can be a function
returned by formatter which is then applied to out-stream and
arg*. Output to out-string, out-stream or, if first argument
is T, to *standard-output*. Return nil. If first argument
is nil, return formatted output.

[min-col] [col-trunc] [min-pad] [pad-char]] [\|] [\|A\|S]

> Print argument of any type for consumption by humans/by the reader, respectively. With \|, print nil as () rather than nil; with A, add pad-chars on the left rather than on the right.
9.4 Macros

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

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

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

(&rest \[var\] (macro-λ*)) \[E\] )

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

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

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

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

One toplevel \[E\] may be replaced by &environment var where var carries the lexical compilation environment. supplied-p is T if there is no corresponding argument.

\(\{\text{defmacro} \{\text{define-compiler-macro}\} \{\text{foo} \{\text{setf} \text{foo}\}\} \{\text{macro-λ*}\} \{\text{declare decl}^*\}\}^k\)

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{define} \text{symbol-macro} \{\text{foo} \text{form}\}\)\

\(\{\text{macro-λ*}\} \{\text{declare} \text{decl}^*\}\)\

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{macro} \text{form}\}^k\)\

\(\{\text{define} \text{symbol-macro} \{\text{foo} \text{form}\}\)\

\(\{\text{macro-λ*}\} \{\text{declare} \text{decl}^*\}\)\

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{macro} \text{form}\}^k\)\

\(\{\text{define} \text{symbol-macro} \{\text{foo} \text{form}\}\)\

\(\{\text{macro-λ*}\} \{\text{declare} \text{decl}^*\}\)\

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{macro} \text{form}\}^k\)\n
One toplevel \[E\] may be replaced by &environment var where var carries the lexical compilation environment. supplied-p is T if there is no corresponding argument.

\(\{\text{defmacro} \{\text{define-compiler-macro}\} \{\text{foo} \{\text{setf} \text{foo}\}\} \{\text{macro-λ*}\} \{\text{declare decl}^*\}\}^k\)

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{define} \text{symbol-macro} \{\text{foo} \text{form}\}\)

\(\{\text{macro-λ*}\} \{\text{declare} \text{decl}^*\}\)\

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{macro} \text{form}\}^k\)\

\(\{\text{define} \text{symbol-macro} \{\text{foo} \text{form}\}\)

\(\{\text{macro-λ*}\} \{\text{declare} \text{decl}^*\}\)\

\(\{\text{doc} \text{form}\}^k\)\

\(\{\text{macro} \text{form}\}^k\)\n
One toplevel \[E\] may be replaced by &environment var where var carries the lexical compilation environment. supplied-p is T if there is no corresponding argument.
9.5 Control Flow

- `if (test then [else])`:
  - Return values of `then` if `test` returns `T`; return values of `else` otherwise.

- `cond (test then)`:
  - Return the values of the first `then` whose `test` returns `T`; return `NIL` if all tests return `NIL`.

- `when (test foo)`:
  - Evaluate `foo` and return their values if `test` returns `T` or `NIL`, respectively. Return `NIL` otherwise.

- `case test (keys foo)`:
  - Return the values of the first `foo` one of whose `keys` is `eql` `test`. Return `NIL` if no element of keys matches.

- `case (keys foo)`:
  - Return the values of the first `foo` one of whose `keys` is `eql` `test`. Signal non-correctable/correctable `type-error` and return `NIL` if no element of keys matches.

- `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 `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-prog1 form-r form`:
  - Evaluate `forms` in order. Return values/1st value, respectively, of `form-r`.

- `define-modify-macro foo ([&optional var (var [inst-supplied-sup])])
  - Define macro `foo` able to modify a place. On invocation of `(foo place arg-vars)`, the value of `function` applied to `place` and `arg-vars` will be stored into place and returned.
(readtable-case readable)[macro]
  ▷ Case sensitivity attribute (one of :upcase, :downcase, :preserve, :invert) of readable. settable.

(copy-readable [from-readable readtable to-readable][macro])
  ▷ Return copy of from-readable.

(set-syntax-from-char to-char from-char)[macro]
  ▷ Copy syntax of from-char to to-char. Return T.

*readable* ▷ Current readable.

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

*read-default-float-format*[macro]
  ▷ Floating point format to use when not indicated in the number read.

*read-suppress*[macro]
  ▷ If T, reader is syntactically more tolerant.

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

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

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

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

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

12.3 Macro Characters and Escapes

 multi-line-comment* | #
  ; one-line-comment*

  ▷ Comments. There are conventions:

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

  ( )  ▷ Initiate reading of a list.

  *  ▷ Begin and end of a string.

  'foo  ▷ (quote foo); foo unevaluated

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

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

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

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

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

  #nAsequence  ▷ n-dimensional array.
(doalist (var list result)) (declare decl+ {tag|form}+)
▷ Evaluate tagbody-like body with var successively bound to the elements of list. Upon evaluation of result, var is NIL. Implicitly, the whole form is a block named NIL.

9.7 Loop Facility

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

(loop form+)
▷ Loop Facility. For Loop Facility keywords see below and Figure 1.

declare decl+ {tag|form}+
▷ Give loop’s implicit block a name.

(with {var-s} { {var-s} } [d-type] = foo)+
{and {var-p} { {var-p} } [d-type] = bar}+
where destructuring type specifier d-type has the form
{fixnum|float|NIL|of-type {type} {type}+)
▷ Initialize (possibly trees of) local variables var-s sequentially and var-p in parallel.

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

{foras} {var-s} {d-type}+ {and {var-p} { {var-p} } [d-type] }+
▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables var-s sequentially and var-p in parallel. Destructuring type specifier d-type as with with.

(upfrom|from|downfrom) start
▷ Start stepping with start

to|downto|above|below|form
▷ Specify form as the end value for stepping.

{in|on} list
▷ Bind var to successive elements/fields, respectively, of list.

by {step|function|fn}
▷ Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.

= foo {then|when|when|then|then} bar
▷ Bind var in the first iteration to foo and later to bar.

across vector
▷ Bind var to successive elements of vector.

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

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

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

{symbol|symbols} {present-symbol|present-symbols} external-symbol {external-symbols} {of|in}
package
class
▷ 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.

it
▷ Value of test form of an enclosing if, when, or unless clause.

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

(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.)

12.2 Reader

{yes-or-no-p|no-p} [control arg]*
▷ Ask user a question and return T or NIL depending on their answer. See p. 35, format, for control and args.

(with-standard-io-syntax form+)
▷ Evaluate forms with standard behaviour of reader and printer. Return values of forms.

(read-preserving-whitespace stream) [stream standard-input] [eof-error \[EOF\]]
▷ Read printed representation of object.

(read-from-string string) [eof-error \[EOF\]]
▷ Read a string from and return object read from string and zero-indexed position of next character.

read-delimited-list char stream standard-input [recursive t]
▷ Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.

(read-char -stream standard-input \[EOF\] [eof-value \[NIL\]] [recursively t])
▷ Return next character from stream.

(read-char-no-hang -stream \[EOF\] [eof-value \[NIL\]] [recursively t])
▷ Next character from stream or NIL if none is available.

(peek-char mode stream standard-input [eof-error \[EOF\]] [eof-value \[NIL\]] [recursively t])
▷ 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)
▷ Put last read character back into stream; return NIL.

(read-byte stream \[EOF\] [eof-value \[NIL\]])
▷ Read next byte from binary stream.

(read-line stream standard-input \[EOF\] [eof-value \[NIL\]] [recursively t])
▷ Return a line of text from stream and T if line has been ended by end of file.

(read-sequence sequence stream \[start start\]|end end)
▷ Replace elements of sequence between start and end with elements from stream. Return index of sequence’s first unmodified element.
12 Input/Output

12.1 Predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(streamp foo)</td>
<td>T if foo is of indicated type.</td>
</tr>
<tr>
<td>(pathnamep foo)</td>
<td></td>
</tr>
<tr>
<td>(readablep foo)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Loop Facility, Overview.

Figure 2: Condition Types.

† For supertypes of this type look for the instance without a †.
\( \text{return \{form\|it\} } \)
\( \triangleright \) Return immediately, skipping any \text{finally} parts, with values of \text{form} or it.

\( \text{collect\{collecting\} \{form\|it\} \text{[into list]} } \)
\( \triangleright \) Collect values of \text{form} or it into list. If no list is given, collect into an anonymous list which is returned after termination.

\( \text{append\{appending\} nconc\{nconc\} \{form\|it\} \text{[into list]} } \)
\( \triangleright \) Concatenate values of \text{form} or it, which should be lists, into list by means of \text{append} or \text{nconc}, respectively. If no list is given, collect into an anonymous list which is returned after termination.

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

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

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

\( \text{if\{when\} unless\{unless\} test\{i-form\} and\{j-form\}*[else\{k-form\} and\{l-form\}*[end]\{\} ] } \)
\( \triangleright \) If \text{test} returns \text{T}, \text{T}, or \text{NIL}, respectively, evaluate \text{i-form} and \text{j-forms}; otherwise, evaluate \text{k-form} and \text{l-forms}. Inside \text{i-form} and \text{k-form}, the value of \text{test} is accessible by it.

\( \text{repeat\{num\} \text{[num]} } \)
\( \triangleright \) Terminate \text{loop} after \text{num} iterations; \text{num} is evaluated once.

\( \text{while\{until\} test \{\} } \)
\( \triangleright \) Continue iteration until \text{test} returns \text{NIL} or \text{T}, respectively.

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

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

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

10 CLOS

10.1 Classes

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

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

\( \text{defclass foo \{superclass\} standard-object\{\} } \)
\( \{\text{slot}\} \)
\( \{\text{reader reader-function}\} \)
\( \{\text{writer writer-function}\} \)
\( \{\text{accessor reader-function}\} \)
\( \{\text{allocation}\{\text{instance}\} \{\text{instance}\} \{\text{instance}\} \} \)
\( \{\text{initarg}\{\text{initarg-name}\}\{\text{initarg}\}\{\text{initarg-name}\} \}\)
\( \{\text{initform}\ \text{form}\} \)
\( \{\text{type}\ \text{type}\} \)
\( \{\text{documentation}\ \text{slot-doc}\} \)
11 Conditions and Errors

(define-condition foo (parent-type condition)

(slot {reader reader} {writer writer} {accessor reader} {allocation instance} {initarg :initarg-name} {infoform form} {type type} {documentation slot-document} {default-initials (name value)} {documentation condition-document} {report string} {report-function}))

» Define, as a subtype of parent-types, condition type foo. In new conditions, a slot’s value defaults to form unless set via :initarg-name, and is accessible by function reader and by generic function writer. With :allocation :class, slot is shared by all conditions of type foo. A condition is reported by string or by report-function of arguments condition and stream.

(make-condition type (initarg-name value))

» Return new condition of type.

(signal warn error)

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

(error continue-control)

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

(ignore-errors form)

» Return values of forms or, in case of error, NIL and the condition.

:invoke-debugger condition)

» Invoke debugger with condition.

(assert test (place) (type (initarg-name value)) (control arg))

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

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

» Return class named symbol. setfable.

(make-instance class (initarg value))

» Make new instance of class.

(reinitialize-instance instance (initarg value))

» Change local slots of instance according to initargs.

(slot-value foo slot)

» Return value of slot in foo. setfable.

(slot-makunbound instance slot)

» Make slot in instance, unbound.

(with-slots (slot (car slot)) (other-keyarg) (instance (declare decl)* form))

» Return values of forms after evaluating them in a lexical environment with slots of instance visible as setfable slots or vars/with accessors of instance visible as setfable vars.

(class-name class)

{setf class-name} new-name class)

» Get/set name of class.

(class-of foo)

» Class foo is a direct instance of.

(change-class instance new-class (initarg value))

» Change class of instance to new-class.

(make-instances-obselete class)

» Update instances of class.

(initialize-instance (instance)

{initialize-instance-for-different-class} previous current)

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

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

» Its primary method sets slots on behalf of make-instances-obselete by means of shared-initialize.

(allocate-instance class (initarg value))

» Return uninitialized instance of class. Called by make-instance.

(shared-initialize instance slots (initarg value))

» Fill instance’s slots using initargs and :initform forms.

(slot-missing class object slot)

(setf slot-boundp slot-makunbound slot-value)

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

(slot-unbound class instance slot)

» Called by slot-value in case of unbound slot. Its primary method signals unbound-slot.
10.2 Generic Functions

- **(next-method)** □ If enclosing method has a next method.

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

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

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

▷ Define or modify generic function foo.

- **(defmethod)** {foo (setf foo)} {before after around qualifier*}

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

- **(add-method)** generic-function method

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

- **(remove-method)** generic-function method

▷ Return suitable method, or signal error.

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

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

- **(call-next-method)** arg*

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

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

▷ Called on invocation of generic-function on args if there is no applicable method. Default method signals error.

10.3 Method Combination Types

- **(standard)**

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

- **(and-or-append-list-nconc-string-max-min)**

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

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

▷ 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, have generic function applied to gen-arg* return with the values of (c-type {primary-method gen-arg*})", leastmost primary-method being the most specific. In **defmethod**, primary methods are denoted by the qualifier c-type.

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

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