Quick Reference

cl

Common lisp

Bert Burgemeister
Common Lisp Quick Reference

Contents

1 Numbers ................................. 3
1.1 Predicates ............................. 3
1.2 Numeric Functions ................. 3
1.3 Logic Functions ..................... 4
1.4 Integer Functions ................... 5
1.5 Implementation-Dependent ........ 6
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 Sequence Predicates ............... 12
6.2 Sequence 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 .................................. 23
10.1 Classes ............................... 23
10.2 Generic Functions ................. 25
10.3 Method Combination 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 Pathnames and Files .............. 40
14 Packages and Symbols .............. 42
14.1 Predicates ............................ 42
14.2 Packages ............................. 42
14.3 Symbols .............................. 43
14.4 Standard Packages ................. 44
15 Compiler ................................ 44
15.1 Predicates ............................ 44
15.2 Compilation .......................... 44
15.3 REPL and Debugging .............. 45
15.4 Declarations ......................... 46
16 External Environment ............... 47

Typographic Conventions

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

\( f = \text{number} \Rightarrow \) if all numbers, or none, respectively, are equal in value.

\( f \neq \text{number} \Rightarrow \) if any numbers are non-equal in value.

\( f > \text{number} \Rightarrow \) return \( T \) if numbers are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

\( \text{minusp} \ a \Rightarrow T \) if \( a < 0 \), otherwise \( T \).

\( \text{zerop} \ a \Rightarrow T \) if \( a = 0 \), otherwise \( T \).

\( \text{plusp} \ a \Rightarrow T \) if \( a > 0 \), otherwise \( T \).

\( \text{evenp} \ \text{int} \Rightarrow T \) if \( \text{int} \) is even, otherwise \( T \).

\( \text{oddp} \ \text{int} \Rightarrow T \) if \( \text{int} \) is odd, otherwise \( T \).

\( \text{numberp} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is of indicated type.

\( \text{realp} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is a real number.

\( \text{rationalp} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is a rational number.

\( \text{floatp} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is a floating-point number.

\( \text{integerp} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is an integer.

\( \text{complexp} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is a complex number.

\( \text{random-state-p} \ \text{foo} \Rightarrow T \) if \( \text{foo} \) is a random state.

1.2 Numeric Functions

\( + a 0 \Rightarrow \sum a \) or \( \prod a \), respectively.

\( \times a b \Rightarrow a \times b \) or \( a \times \prod b \), respectively. Without any \( b \)s, return 0 or \( 1/a \), respectively.

\( 1+ a \Rightarrow a + 1 \) or \( a - 1 \), respectively.

\( 1- a \Rightarrow a - 1 \) or \( a + 1 \), respectively.

\( \{ \text{incf} \ \text{place} [\text{delta}] \} \Rightarrow \text{Increment or decrement the value of } \text{place} \text{ by } \text{delta}. \text{ Return new value.} \)

\( \exp p \Rightarrow e^p \) or \( b^p \), respectively.

\( \log a \ [b] \Rightarrow \log_{b} a \) or, without \( b \), \( \ln a \).

\( \sqrt[\text{n}] n \Rightarrow \sqrt[\text{n}] a \) in complex numbers/natural numbers.

\( \lcm \text{integer} \ [\text{integer}] \Rightarrow \text{Least common multiple or greatest common denominator, respectively, of integers. (gcd) returns 1.} \)

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

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

\( \text{asin} \ \text{a} \Rightarrow \arcsin \text{a} \) or \( \arccos \text{a} \), respectively, in radians.

\( \text{atan} \ \text{a} \Rightarrow \arctan \text{a} \), \( \text{atan2} \ \text{a} \Rightarrow \arctan \text{a} \), respectively, in radians.

\( \text{sinh a} \Rightarrow \sinh a \), \( \cosh a \Rightarrow \cosh a \), \( \tanh a \Rightarrow \tanh a \), respectively.
\(\text{asinh} a\) \(\text{acosh} a\) \(\text{atanh} a\)

- Return \(\sqrt{a^2 + 1}\) for \(a \geq 0\), \(\sqrt{a^2 - 1}\) for \(a < 0\).
- Return complex conjugate of \(a\).
- Return as integer or float, respectively, \(n/d\) rounded, or rounded towards \(-\infty, +\infty, 0\), respectively; and remainder.
- Return non-negative random number less than \(\text{limit}\), and of the same type.
- Current random state.
- Number of magnitude 1 representing sign or phase of \(n\).
- Numerator or denominator, respectively, of \(\text{rational}\)'s canonical form.
- Real part or imaginary part, respectively, of \(\text{number}\).
- Make a complex number.
- Angle of \(\text{number}\)'s polar representation.
- Return \(|n|\).
- Convert real to rational. Assume complete/limited accuracy for \(\text{real}\).
- Convert real into float with type of \(\text{float}\) prototype.

1.3 Logic Functions

Negative integers are used in two's complement representation.

(\text{boole operation int-a int-b})
- Return value of bitwise logical \(\text{operation}\). \(\text{operations}\) are
  - \text{boole-1} \(\Rightarrow\) \(\text{int-a}\).
  - \text{boole-2} \(\Rightarrow\) \(\text{int-b}\).
  - \text{boole-c1} \(\Rightarrow\) \(\neg\text{int-a}\).
  - \text{boole-c2} \(\Rightarrow\) \(\neg\text{int-b}\).
  - \text{boole-set} \(\Rightarrow\) All bits set.
  - \text{boole-clr} \(\Rightarrow\) All bits zero.
Common Lisp Quick Reference

\[ \text{boole-eqv} \triangleq \text{int}-a \equiv \text{int}-b, \]
\[ \text{boole-and} \triangleq \text{int}-a \land \text{int}-b, \]
\[ \text{boole-andc1} \triangleq \neg \text{int}-a \land \text{int}-b, \]
\[ \text{boole-andc2} \triangleq \text{int}-a \neg \land \text{int}-b, \]
\[ \text{boole-nand} \triangleq \neg (\text{int}-a \land \text{int}-b), \]
\[ \text{boole-ior} \triangleq \text{int}-a \lor \text{int}-b, \]
\[ \text{boole-orc1} \triangleq \neg \text{int}-a \lor \text{int}-b, \]
\[ \text{boole-orc2} \triangleq \text{int}-a \lor \neg \text{int}-b, \]
\[ \text{boole-xor} \triangleq \neg (\text{int}-a \equiv \text{int}-b), \]
\[ \text{boole-nor} \triangleq \neg (\text{int}-a \lor \text{int}-b). \]

\( (\text{f lognot integer}) \triangleq \neg \text{integer}. \)
\( (\text{f logeqv integer*}) \)
\( (\text{f logand integer*}) \)
\( \triangleright \text{Return value of exclusive-nored or anded integers, respectively. Without any integer, return -1.} \)
\( (\text{f logandc1 int-a int-b}) \)
\( (\text{f logandc2 int-a int-b}) \)
\( (\text{f lognand int-a int-b}) \)
\( (\text{f logxor integer*}) \)
\( (\text{f logior integer*}) \)
\( \triangleright \text{Return value of exclusive-ored or ored integers, respectively. Without any integer, return 0.} \)
\( (\text{f logorc1 int-a int-b}) \)
\( (\text{f logorc2 int-a int-b}) \)
\( (\text{f lognor int-a int-b}) \)
\( (\text{f logbitp i int}) \quad \triangleright \text{T if zero-indexed } i \text{th bit of } \text{int} \text{ is set.} \)
\( (\text{f logtest int-a int-b}) \quad \triangleright \text{Return } \text{T} \text{ if any bit set in } \text{int-a} \text{ which is set in } \text{int-b} \text{ as well.} \)
\( (\text{f logcount int}) \quad \triangleright \text{Number of 1 bits in } \text{int} \geq 0, \text{ number of 0 bits in } \text{int} < 0. \)

1.4 Integer Functions

\( (\text{f integer-length integer}) \quad \triangleright \text{Number of bits necessary to represent } \text{integer}. \)
\( (\text{f ldb-test byte-spec integer}) \quad \triangleright \text{Return } \text{T} \text{ if any bit specified by } \text{byte-spec in } \text{integer} \text{ is set.} \)
\( (\text{f ash integer count}) \quad \triangleright \text{Return copy of } \text{integer} \text{ arithmetically shifted left by } \text{count} \text{ adding zeros at the right, or, for } \text{count} < 0, \text{ shifted right discarding bits.} \)
\( (\text{f ldb byte-spec integer}) \quad \triangleright \text{Extract } \text{byte} \text{ denoted by } \text{byte-spec from } \text{integer}. \text{ setfable.} \)
\( (\text{f deposit-field}) \quad \{ \text{f dpb} \} \quad \text{int-a byte-spec int-b} \quad \triangleright \text{Return } \text{int-b} \text{ with bits denoted by } \text{byte-spec} \text{ replaced by corresponding bits of } \text{int-a}, \text{ or by the low } (\text{f byte-size byte-spec}) \text{ bits of } \text{int-a}, \text{ respectively.} \)
\( (\text{f mask-field byte-spec integer}) \quad \triangleright \text{Return copy of } \text{integer} \text{ with all bits unset but those denoted by } \text{byte-spec}. \text{ setfable.} \)
\( (\text{f byte size position}) \quad \triangleright \text{Byte specifier for a byte of size } \text{bits starting at a weight of } 2^{\text{position}}. \)
\( (\text{f byte-size byte-spec}) \quad \triangleright \text{Size or position, respectively, of } \text{byte-spec}. \)
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

▷ Smallest possible number making a difference when added or subtracted, respectively.

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

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

▷ Available numbers closest to −0 or +0, respectively.

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

- fixnum

▷ Available numbers closest to −∞ or +∞, respectively.

(decode-float n)
(integer-decode-float n)
(scale-float n i)
(float-radix n)
(float-digits n)
(float-precision n)
(upgraded-complex-part-type foo [environment NIL])

▷ Return significand, exponent, and sign of float n.

With n’s radix b, return nb^i.

▷ Type of most specialized complex number able to hold parts of type foo.

2 Characters

The standard-char type comprises a-z, A-Z, 0-9, Newline, Space, and !?$'*.:;=-<>\%&@#|}\{^<=>

(characterp foo)
(standard-char-p char)
(graphic-char-p character)
(alpha-char-p character)
(alphanumericp character)

▷ T if argument is of indicated type.

(upper-case-p character)
(lower-case-p character)
(both-case-p character)

▷ Return T if character is uppercase, lowercase, 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+)
(char/= character+)
(char=equal character+)
(char-not-equal character+)

▷ Return T if all characters, or none, respectively, are equal.

(char>< character+)
(char>= character+)
(char<= character+)

▷ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.
3 Strings

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

(defun foo (x) ; ...)
(defun simple-string-p foo) ; ...
(defun string= foo) ; ...
(defun string-not= foo) ; ...
(defun string= string-not=) ; ...
(defun make-string size) ; ...
(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(defun nstring-capitalize x) ; ...
(defun nstring-upcase x) ; ...
(defun nstring-downcase x) ; ...
(defun string-trim x) ; ...
(defun string-left-trim x) ; ...
(defun string-right-trim x) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...
(defun char-bag string) ; ...

(defun string x) ; ...
(defun string-capitalize x) ; ...
(defun string-upcase x) ; ...
(defun string-downcase x) ; ...
(defun nstring x) ; ...
(de...
4 Conses

4.1 Predicates

(ccons foo)  ➞ Return T if foo is of indicated type.
(cnendp list)  ➞ Return T if list/foo is NIL.
(cnull foo)  ➞ Return T if foo is not a cons.
(ctrailp foo list)  ➞ Return T if foo is a tail of list.

(member-foo-list test function key function)  ➞ Return tail of list starting with its first element matching foo. Return NIL if there is no such element.

(member-if test function key function)  ➞ Return tail of list starting with its first element satisfying test. Return NIL if there is no such element.

(subsetp list-a list-b test function test-not function key function)  ➞ Return T if list-a is a subset of list-b.

4.2 Lists

(ccons foo bar)  ➞ Return new cons (foo . bar).
(clist foo+)  ➞ Return list of foos.
(clist+ foo+)  ➞ Return list of foos with last foo becoming cdr of last cons. Return foo if only one foo given.

(make-list num [initial-element foo])  ➞ New list with num elements set to foo.
(clist-length list)  ➞ Length of list; NIL for circular list.
(car list)  ➞ Car of list or NIL if list is NIL. setfable.
(cdr list)  ➞ Cdr of list or NIL if list is NIL. setfable.
(cdr cdr n list)  ➞ Return tail of list after calling cdr n times.

([first second third fourth fifth sixth . . ninth tenth] list)  ➞ Return nth element of list if any, or NIL otherwise. setfable.

(nth n list)  ➞ Zero-indexed nth element of list. setfable.

(cXr list)  ➞ With X being one to four as and ds representing 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.
\((\text{butlast } \text{list}) \text{ [num]}\) \> list excluding last num conses.

\((\text{nbutlast } \text{list})\)
\> Replace car, or cdr, respectively, of cons with object.

\((\text{rdiff } \text{foo})\)
\> If foo is a tail of list, return preceding part of list. Otherwise return list.

\((\text{adjoin } \text{foo } \text{list})\)
\> Return list if foo is already member of list. If not, return \((\text{cons } \text{foo} \text{list})\).

\((\text{pop } \text{place})\)
\> Set place to \((\text{cdr } \text{place})\), return \((\text{car } \text{place})\).

\((\text{push } \text{foo } \text{place})\)
\> Set place to \((\text{cons } \text{foo} \text{place})\).

\((\text{pushnew } \text{foo } \text{place})\)
\> Set place to \((\text{adjoin } \text{foo} \text{place})\).

\((\text{append } \text{proper-list}^\ast \text{foo} \text{NIL})\)
\> Return concatenated list or, with only one argument, foo can be of any type.

\((\text{revappend } \text{foo} \text{list})\)
\> Return concatenated list after reversing order in list.

\((\text{mapcar} \text{function } \text{list}^\ast)\)
\> Return list of return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list.

\((\text{mapcan} \text{function } \text{list}^\ast)\)
\> Return list of concatenated return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

\((\text{mapc} \text{function } \text{list}^\ast)\)
\> Return first list after successively applying function to corresponding arguments, either cars or cdrs, respectively, from each list. function should have some side effects.

\((\text{copy-list } \text{list})\)
\> Return copy of list with shared elements.

4.3 Association Lists

\((\text{pairlis } \text{keys } \text{values} \text{alist}^\ast)\)
\> Prepend to alist an association list made from lists keys and values.

\((\text{acons } \text{key } \text{value} \text{alist})\)
\> Return alist with a (key , value) pair added.

\((\text{assoc} \text{foo} \text{alist})\)
\> First cons whose car, or cdr, respectively, satisfies test.

\((\text{copy-alist } \text{alist})\)
\> Return copy of alist.
4.4 Trees

(tree-equal foo bar ([:test test eql])
  \> Return T if trees foo and bar have same shape and leaves satisfying Test.

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

(row-major-aref array i)
  \> Return ith element of array in row-major order. setfable.

4.5 Sets

(intersection set-exclusive-or set-exclusive-or)
  \> Return a \cap b, a \setminus b, a \cup b, or a \triangle b, respectively, of lists a and b.

5 Arrays

5.1 Predicates

(arrayp foo)
  \> T if foo is of indicated type.

5.2 Array Functions

(make-array dimension-sizes
  \> Return fresh, or readjust, respectively, vector or array.

(setf
  \> Return array element pointed to by subscripts. setfable.

(row-major-aref array i)
  \> Return ith element of array in row-major order. setfable.
\texttt{array-row-major-index array [subscripts]}
\begin{itemize}
\itemchip Index in row-major order of the element denoted by \texttt{subscripts}.
\end{itemize}

\texttt{array-dimensions array}
\begin{itemize}
\itemchip List containing the lengths of \texttt{array}'s dimensions.
\end{itemize}

\texttt{array-dimension array i}
\begin{itemize}
\itemchip Length of \texttt{i}th dimension of \texttt{array}.
\end{itemize}

\texttt{array-total-size array}
\begin{itemize}
\itemchip Number of elements in \texttt{array}.
\end{itemize}

\texttt{array-rank array}
\begin{itemize}
\itemchip Number of dimensions of \texttt{array}.
\end{itemize}

\texttt{array-displacement array}
\begin{itemize}
\itemchip Target array and offset.
\end{itemize}

\texttt{bit bit-array [subscripts]}
\begin{itemize}
\itemchip Return element of \texttt{bit-array} or of \texttt{simple-bit-array}. \texttt{setf}-able.
\end{itemize}

\texttt{bit-not bit-array [result-bit-array NIL]}
\begin{itemize}
\itemchip Return result of bitwise negation of \texttt{bit-array}. If \texttt{result-bit-array} is \texttt{T}, put result in \texttt{bit-array}; if it is \texttt{NIL}, make a new array for result.
\end{itemize}

\begin{itemize}
\itemchip \texttt{bit-eqv} \texttt{bit-and} \texttt{bit-andc1} \texttt{bit-andc2} \texttt{bit-nand} \texttt{bit-ior} \texttt{bit-orc1} \texttt{bit-orc2} \texttt{bit-xor} \texttt{bit-nor}
\end{itemize}
\begin{itemize}
\itemchip Return result of bitwise logical operations (cf. operations of \texttt{boole}, page 4) on \texttt{bit-array-a} and \texttt{bit-array-b}. If \texttt{result-bit-array} is \texttt{T}, put result in \texttt{bit-array-a}; if it is \texttt{NIL}, make a new array for result.
\end{itemize}

\texttt{array-rank-limit}
\begin{itemize}
\itemchip Upper bound of array rank; \geq 8.
\end{itemize}

\texttt{array-dimension-limit}
\begin{itemize}
\itemchip Upper bound of an array dimension; \geq 1024.
\end{itemize}

\texttt{array-total-size-limit}
\begin{itemize}
\itemchip Upper bound of array size; \geq 1024.
\end{itemize}

\subsection*{5.3 Vector Functions}
Vectors can as well be manipulated by sequence functions; see section 6.

\texttt{vector \texttt{foo}^*}
\begin{itemize}
\itemchip Return fresh \texttt{simple} vector of \texttt{foo}.
\end{itemize}

\texttt{svref vector i}
\begin{itemize}
\itemchip Element \texttt{i} of \texttt{simple vector}. \texttt{setf}-able.
\end{itemize}

\texttt{vector-push \texttt{foo} \texttt{vector}^*}
\begin{itemize}
\itemchip Return \texttt{NIL} if \texttt{vector}'s fill pointer equals size of \texttt{vector}. Otherwise replace element of \texttt{vector} pointed to by fill pointer with \texttt{foo}; then increment fill pointer.
\end{itemize}

\texttt{vector-push-extend \texttt{foo} \texttt{vector} [num]}
\begin{itemize}
\itemchip Replace element of \texttt{vector} pointed to by fill pointer with \texttt{foo}, then increment fill pointer. \texttt{Extend vector}'s size by \geq \texttt{num} if necessary.
\end{itemize}

\texttt{vector-pop \texttt{vector}^*}
\begin{itemize}
\itemchip Return element of \texttt{vector} its fill pointer points to after decrementation.
\end{itemize}

\texttt{fill-pointer vector}
\begin{itemize}
\itemchip Fill pointer of \texttt{vector}. \texttt{setf}-able.
6 Sequences

6.1 Sequence Predicates

\[
\begin{aligned}
\{ \text{every} \} & \quad \text{test sequence}^+ \\
\{ \text{notevery} \} & \quad \text{test sequence}^+
\end{aligned}
\]

▷ Return \texttt{NIL} or \texttt{T}, respectively, as soon as \texttt{test} on any set of corresponding elements of \texttt{sequences} returns \texttt{NIL}.

\[
\begin{aligned}
\{ \text{some} \} & \quad \text{test sequence}^+ \\
\{ \text{notany} \} & \quad \text{test sequence}^+
\end{aligned}
\]

▷ 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}.

\[
\{ \text{mismatch} \} \quad \text{sequence-a sequence-b}
\]

▷ 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

\[
\begin{aligned}
\{ \text{make-sequence} \} & \quad \text{sequence-type size [initial-element foo]} \\
\{ \text{concatenate} \} & \quad \text{type sequence}^+
\end{aligned}
\]

▷ Make sequence of \texttt{sequence-type} with size elements.

\[
\begin{aligned}
\{ \text{merge} \} & \quad \text{type-a sequence-a sequence-b test [key function set]} \\
\{ \text{fill} \} & \quad \text{sequence foo [start start] [end end]}
\end{aligned}
\]

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

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

\[
\begin{aligned}
\{ \text{length} \} & \quad \text{sequence} \\
\{ \text{count} \} & \quad \text{foo sequence}
\end{aligned}
\]

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

▷ Return number of elements in \texttt{sequence} which match \texttt{foo}.

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

▷ Return number of elements in \texttt{sequence} which satisfy \texttt{test}.

\[
\begin{aligned}
\{ \text{elt} \} & \quad \text{sequence index} \\
\{ \text{subseq} \} & \quad \text{sequence start [end]} \\
\{ \text{sort} \} & \quad \text{sequence test [key function]}
\end{aligned}
\]

▷ Return element of \texttt{sequence} pointed to by zero-indexed \texttt{index}. \texttt{setf}able.

▷ Return \texttt{subsequence} of \texttt{sequence} between \texttt{start} and \texttt{end}. \texttt{setf}able.

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

\[
\begin{aligned}
\{ \text{reverse} \} & \quad \text{sequence} \\
\{ \text{reverses} \} & \quad \text{sequence}
\end{aligned}
\]

▷ Return \texttt{sequence} in reverse order.

12

Common Lisp Quick Reference
(\(\text{return-first-element-in-sequence}\) \(\text{which}\) \(\text{matches}\) \(\text{foo}\), \(\text{or}\) its \(\text{position}\) relative to the begin of \(\text{sequence}\), respectively.

\(\text{return-first-element-in-sequence}\) \(\text{which}\) \(\text{satisfies}\) \(\text{test}\), \(\text{or}\) its \(\text{position}\) relative to the begin of \(\text{sequence}\), respectively.

\(\text{search-sequence-a-sequence-b}\) \(\text{sequence-a}\) \(\text{sequence-b}\). \(\text{Return}\) \(\text{position-in-sequence-b}\), \(\text{or}\) \(\text{NIL}\).

\(\text{remove-foo-sequence}\) \(\text{delete-foo-sequence}\) \(\text{Make}\) \(\text{copy}\) \(\text{of}\) \(\text{sequence}\) \(\text{without}\) \(\text{elements}\) \(\text{satisfying}\) \(\text{test}\) removed.

\(\text{remove-duplicates-sequence}\) \(\text{delete-duplicates-sequence}\) \(\text{Make}\) \(\text{copy}\) \(\text{of}\) \(\text{sequence}\) \(\text{without}\) \(\text{duplicates}\).

\(\text{substitute}\) \(\text{new}\) \(\text{old}\) \(\text{sequence}\) \(\text{substitute}\) \(\text{new}\) \(\text{old}\) \(\text{sequence}\) \(\text{Make}\) \(\text{copy}\) \(\text{of}\) \(\text{sequence}\) \(\text{with}\) \(\text{all}\) \(\text{(or}\) \(\text{count}\) \(\text{elements}\) \(\text{satisfying}\) \(\text{test}\) replaced by \(\text{new}\).

\(\text{substitute-if}\) \(\text{substitute-if-not}\) \(\text{substitute-if-not}\) \(\text{substitute-if-not}\) \(\text{substitute-if-not}\) \(\text{substitute-if-not}\) \(\text{Replace}\) \(\text{elements}\) \(\text{of}\) \(\text{sequence-a}\) \(\text{with}\) \(\text{elements}\) \(\text{of}\) \(\text{sequence-b}\).
(\texttt{map \textit{type function sequence}^+})
\textbullet\hspace{1em} \text{Apply function successively to corresponding elements of the sequences. Return values as a sequence of type. If type is NIL, return NIL.}

(\texttt{map-into \textit{result-sequence function sequence}^*})
\textbullet\hspace{1em} \text{Store into result-sequence successively values of function applied to corresponding elements of the sequences.}

(\texttt{reduce \textit{function sequence}})
\textbullet\hspace{1em} \text{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.}

(\texttt{copy-seq \textit{sequence}})
\textbullet\hspace{1em} \text{Copy of sequence with shared elements.}

7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see \texttt{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.

(\texttt{hash-table-p \textit{foo}})
\textbullet\hspace{1em} \text{Return T if \textit{foo} is of type hash-table.}

(\texttt{make-hash-table \{\texttt{test \{eq\|eq\|equal\|equalp\}}\} \textit{size int \| rehash-size num \| rehash-threshold num}})
\textbullet\hspace{1em} \text{Make a hash table.}

(\texttt{gethash key hash-table [default NIL]})
\textbullet\hspace{1em} \text{Return object with key if any or default otherwise; and T if found, NIL otherwise.}

(\texttt{hash-table-count hash-table})
\textbullet\hspace{1em} \text{Number of entries in hash-table.}

(\texttt{remhash key hash-table})
\textbullet\hspace{1em} \text{Remove from hash-table entry with key and return T if it existed. Return NIL otherwise.}

(\texttt{clrhash hash-table})
\textbullet\hspace{1em} \text{Empty hash-table.}

(\texttt{maphash \textit{function hash-table}})
\textbullet\hspace{1em} \text{Iterate over hash-table calling \textit{function} on key and value. Return NIL.}

(\texttt{with-hash-table-iterator (\textit{foo hash-table}) (\texttt{declare \texttt{\* decl\* \texttt{\* form\*})}})
\textbullet\hspace{1em} \text{Return values of forms. In forms, invocations of (\textit{foo}) return: T if an entry is returned; its key; its value.}

(\texttt{hash-table-test hash-table})
\textbullet\hspace{1em} \text{Test function used in hash-table.}

(\texttt{sxhash \textit{foo}})
\textbullet\hspace{1em} \text{Hash code unique for any argument \textit{eq} \textit{foo}.}
8 Structures

8.1 Predicates

(=f oo bar) ≥ T if foo and bar are identical.

(#eoo bar) ≥ T if foo and bar are identical, or the same character, or numbers of the same type and value.

(#eoo bar) ≥ T if foo and bar are identical; or are the same character ignoring case; or are numbers of the same value ignoring type; or are equivalent pathnames; or are conses or arrays of the same shape with #equalp elements; or are structures of the same type with #equal 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.

(/not moo) ≥ T if moo is NIL; NIL otherwise.

(/bound symbol) ≥ T if symbol is a special variable.

(/constantp moo (environment&rest)) ≥ T if moo is a constant form.

9 Control Structure
\((\text{function} \ foo)\) \implies \text{T} \text{ if } \text{foo} \text{ is of type } \text{function}.

\((\text{fboundp} \ \{\text{foo} \ (\text{setf} \ \text{foo})\})\) \implies \text{T} \text{ if } \text{foo} \text{ is a global function or macro.}

9.2 Variables

\(\{\text{makeconstant} \ \text{foo} [\text{form} [\text{doc}]]\}\)

\implies \text{Assign value of form to global constant/dynamic variable } \text{foo}.

\(\{\text{makeparameter} \ \text{foo} [\text{form} [\text{doc}]]\}\)

\implies \text{Unless bound already, assign value of form to dynamic variable } \text{foo}.

\(\{\text{setf} \ \{\text{setp}\} \ \{\text{place} \ \text{form}\}'\}\)

\implies \text{Set places to primary values of forms. Return values of last form/NIL; work sequentially/in parallel, respectively.}

\(\{\text{setq} \ \{\text{setq}\} \ \{\text{symbol} \ \text{form}\}'\}\)

\implies \text{Set symbols to primary values of forms. Return value of last form/NIL; work sequentially/in parallel, respectively.}

\(\text{setf} \ \text{symbol} \ \text{foo}\)

\implies \text{Set symbol's value cell to } \text{foo}. \text{ Deprecated.}

\(\{\text{remprop} \ \text{symbol} \ \text{key}\}\)

\(\{\text{remf} \ \{\text{place} \ \text{key}\}\}\)

\implies \text{Remove first entry } \text{key} \text{ from property list stored in symbol/place, respectively. Return T if there is no } \text{key}, \text{ setfable.}

\(\{\text{get-properties} \ \text{property-list} \ \text{keys}\}\)

\implies \text{Return key and value of first entry from property-list matching a key from keys}, \text{ and tail of property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.}

\(\{\text{getprop} \ \text{symbol} \ \text{key}\}\)

\(\{\text{rem} \ \text{place} \ \text{key}\}\)

\implies \text{Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.}

\(\{\text{let} \ \{\text{let}\} \ \{\text{name} \ [\text{name} \ [\text{value}\]\text{doc}\]}' \ (\text{declare} \ \text{decl}\) \ (* \ \text{form})*\}\)

\implies \text{Evaluate forms with names lexically bound (in parallel or sequentially, respectively) to values. Return values of forms.}

\(\{\text{multiple-value-bind} \ \{\text{var})* \ \text{values-form} \ (\text{declare} \ \text{decl}\) \ (* \ \text{body-form})*\}\)

\implies \text{Evaluate body-forms with vars lexically bound to the return values of values-form. Return values of body-forms.}
9.3 Functions

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

\[
(\text{var}^\ast \ [\text{optional} \ \text{var} \ \{\text{var} \ \text{init} \ \text{supplied-p}\}] \ [\text{&rest} \ \text{var}])
\]

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

\[
\begin{align*}
(\text{mdefun} \ &\text{foo} \ (\text{ord-}\ast) \\
(\text{mlambda} \ &\text{ord-}\ast) \\
(\text{&let} \ &\text{labels}) \\
(\text{function} \ &\text{foo}) \\
(\text{apply} \ &\text{function}) \\
(\text{funcall} \ &\text{function} \ &\text{arg}^\ast) \\
(\text{values-list} \ &\text{list}) \\
(\text{values} \ &\text{foo}^\ast) \\
(\text{multiple-value-list} \ &\text{form}) \\
(\text{nth-value} \ &\text{n} \ &\text{form}) \\
(\text{complement} \ &\text{function}) \\
(\text{constantly} \ &\text{foo}) \\
(\text{identity} \ &\text{foo})
\end{align*}
\]

\(\triangleright\) Evaluate forms with variables from tree destruct-\*\* bound to corresponding elements of tree bar, and return their values. destruct-\*\* resembles macro-\* (section 9.4), but without any &environment clause.
(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.

(makunbound foo)
  ▷ Remove global function or macro definition foo.

call-arguments-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] (var (macro-λ*)) [E])

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

&rest
  \{rest-var (macro-λ*)\} [E]

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

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

&allow-other-keys
  \{&aux
    \{var \{init [supplied-p]\} \} [E] , rest-var\}.

or

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

One top-level [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.

(define-macro (foo (macro-λ*))
  \{(declare decl*)\* 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 ex-
    panded form.

(define-compiler-macro (foo (macro-λ*))
  \{(declare local-decl*)\* form\*\}

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

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

(defun function (var doc)
  \{(declare decl*) form\*\}

  ▷ Define function with arguments from function.

(defun function (var (s-var) doc)
  \{(declare decl*) form\*\}

  ▷ Define function with arguments from function.

where defsetf lambda list (self-λ*) has the form
(defmacro foo...
  (declare (doc "foo"))
  doc)

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.

(macro-test-fun)
  Evaluate foos and return their values if test returns T or
  NIL, respectively. Return NIL otherwise.
\[\text{(case test (} (\text{key}^a) \text{ foo}^b\text{)} \rightarrow \text{ otherwise} \rightarrow \text{ bar}^n)\]\n
- Return the values of the first \text{ foo}^a one of whose \text{ key}s is \text{ eql} \text{ test}. Return values of \text{ bar} if there is no matching \text{ key}.

\[\text{(ecase test (} (\text{key}^a) \text{ foo}^b\text{)}\text{)}\]

- Return the values of the first \text{ foo}^a one of whose \text{ key}s is \text{ eql} \text{ test}. Signal non-correctable/correctable \text{ type-error} if there is no matching \text{ key}.

\[\text{(and form^e)}\]

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

\[\text{(or form^e)}\]

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

\[\text{(progn form^e)}\]

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

\[\text{(multiple-value-prog1 form-r form^*)}\]

\[\text{(multiple-value-prog2 form-a form-r form^*)}\]

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

\[\text{(prog \text{name} form^e)}\]

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

\[\text{(return-from foo \text{ result}^e)}\]

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

\[\text{(tagbody \{tag\text{form}^f\}}\]

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

\[\text{(go \text{ tag})}\]

- Within the innermost possible enclosing \text{ tagbody}, jump to a tag \text{ eql} \text{ tag}.

\[\text{(catch tag form^e)}\]

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

\[\text{(throw tag form)}\]

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

\[\text{(sleep n)}\]

- Wait \text{ n} seconds; return \text{ NIL}.
9.6 Iteration

\((\text{do} \ m \ \\text{do} \ \^{\text{tag}} \ (\text{stop result}) \ (\text{declare deci}^*\text{)}^* \\text{tagbody} )^* \text{form} )^*\)

▷ Evaluate \text{tagbody}-like body with \text{vars} successively bound according to the values of the corresponding \text{start} and \text{step} forms. \text{vars} are bound in parallel/sequentially, respectively. Stop iteration when \text{stop} is \text{T}. Return values of \text{result}*. Implicitly, the whole form is a \text{block} named \text{NIL}.

\((\text{adotimes} \ m \ i \ \text{result} \text{NIL} )^* \ (\text{declare decl}^*\text{)}^* \text{tagbody} )^*\)

▷ Evaluate \text{tagbody}-like body with \text{var} successively bound to integers from 0 to \text{i} − 1. Upon evaluation of \text{result}, \text{var} is \text{i}. Implicitly, the whole form is a \text{block} named \text{NIL}.

\((\text{adolist} \ \text{list} \ | \text{result} \text{NIL} )^* \ (\text{declare decl}^*\text{)}^* \text{tagbody} )^*\)

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

9.7 Loop Facility

\((\text{loop form}^* )^*\)

▷ Simple Loop. If \text{forms} do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit \text{block} named \text{NIL}.

\((\text{loop clause}^* )^*\)

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

\text{named name} \ \▷ \text{Give \text{loop}'s implicit \text{block} a name.}

\text{with}\ \{\text{var-s}\ \{\text{var-s'}\}^* \) \ \{d-type\} \ [\text{= foo}]^*

\{\text{and}\ \{\text{var-p}\ \{\text{var-p'}\}^* \) \ \{d-type\} \ [\text{= bar}]^*

where destructuring type specifier \text{d-type} has the form \{\text{fixnum float T NIL} \text{| of-type} \{\text{type}\}^*\}

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

\text{for}\ \{\text{var-s}\ \{\text{var-s'}\}^* \) \ \{d-type\}^* \ \{\text{and}\ \{\text{var-p}\ \{\text{var-p'}\}^* \) \ \{d-type\}^*

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

\text{from} \text{start}\ \▷ \text{Start stepping with} \text{start}.

\text{upto} \text{downfrom} \▷ \text{Specify} \text{form} as the end value for stepping.

\text{in} \text{list}\ \▷ \text{Bind} \text{var} to successive elements/tails, respectively, of \text{list}.

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

\text{hash-key} \text{hash-keys}\ \text{of} \text{in} \text{hash-table}\ \text{using}\ \{\text{hash-value} \text{value}\}^*\ \▷ \text{Bind} \text{var} successively to the keys of \text{hash-table}; \text{bind} \text{value} to corresponding \text{values}.

\text{being} \{\text{the}\text{each}\} \ \▷ \text{Iterate over a hash table or a package}.
Figure 1: Loop Facility, Overview.
{hash-value} {hash-values} {of} hash-table [using {hash-key} key]
  ▷ Bind var successively to the values of hash-table; bind key to corresponding keys.
{symbol} {symbols} {present-symbol} {present-symbols} {external-symbol} {external-symbols} {of} {in} package {package}
  ▷ Bind var successively to the accessible symbols, or the present symbols, or the external symbols respectively, of package.

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

10 CLOS

10.1 Classes

{slot-exists-p} foo bar ▷ T if foo has a slot bar.
(slot-boundp instance slot) ⇒ T if slot in instance is bound.

(letf (foo (superclass* standard-object)
  (slot (reader reader) (writer (setf writer))
         (accessor accessor)
         :allocation instance
         :class instance
         :initargs [name value]
         :documentation slot-doc
         :documentation class-doc
         :metaclass name)
  instance)
⇒ 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 [[i]initarg]; it is readable via (reader i) and (accessor i), and writable via (writer value i) or (setf accessor i). slots with :allocation :class are shared by all instances of class foo.

(defun find-class (symbol [errorp [environment]])
⇒ Return class named symbol. setfable.

(defun make-instance-class ([i]initarg-value* other-keyarg*)
⇒ Make new instance of class.

(defun reinitialize-instance instance [i]initarg-value* other-keyarg*)
⇒ Change local slots of instance according to initargs by means of shared-initialize.

(defun slot-value foo slot)
⇒ Return value of slot in foo. setfable.

(defun slot-makunbound instance slot)
⇒ Make slot in instance unbound.

(letf (with-slots ((slot var slot) (var accessor))
         (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 accessor of instance visible as setfable vars.

(defun class-name class)
⇒ Get/set name of class.

(defun class-of foo)
⇒ Class foo is a direct instance of.

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

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

(letf (initialize-instance
tuple-iterate-instance-definition
  previous current)
⇒ Set slots on behalf of make-instance/of change-class by means of shared-initialize.

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

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

(defun class-of foo)
⇒ Class foo is a direct instance of.

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

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

(letf (initialize-instance
tuple-iterate-instance-definition
  previous current)
⇒ Set slots on behalf of make-instance/of change-class by means of shared-initialize.

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

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

(defun class-of foo)
⇒ Class foo is a direct instance of.

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

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

(letf (initialize-instance
tuple-iterate-instance-definition
  previous current)
⇒ Set slots on behalf of make-instance/of change-class by means of shared-initialize.

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

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

(defun class-of foo)
⇒ Class foo is a direct instance of.

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

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

(letf (initialize-instance
tuple-iterate-instance-definition
  previous current)
⇒ Set slots on behalf of make-instance/of change-class by means of shared-initialize.

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

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

(defun class-of foo)
⇒ Class foo is a direct instance of.

(defun change-class instance new-class ([i]initarg-value* other-keyarg*)
⇒ Change class of instance to new-class. Retain the status of any slots that are common between instance's original class and new-class. Initialize any newly added slots with the values of the corresponding initargs if any, or with the values of their :initform forms if not.
10.2 Generic Functions

(next-method-p) ▸ T if enclosing method has a next method.

defgeneric (foo setf foo) (required-var+ &optional var (var))

[argument-precedence-order required-var+] (documentation string)
(generic-function-class gf-class)
(method-class method-class)
(method-combination c-type c-arg*)

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

defmethod (foo setf foo)

[argument-precedence-order required-var+]
(documentation string)
(generic-function-class gf-class)
(method-class method-class)
(method-combination c-type c-arg*)

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

allocate-instance class ([c]omtary value)* other-keyarg*)

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

shared-initialize instance [uniform-slots] ([c]omtary-slot value)*

➤ Fill the instaty-slots of instance with the corresponding values, and fill those uniform-slots that are not instaty-slots with the values of their :uniform forms.

slot-missing class instance slot

[setf slot-boundp]

[setf slot-makunbound]

[setf slot-value]

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

slot-unbound class instance slot

➤ If an instance slot is unbound, it must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to method-class do not propagate to existing methods. For c-type see section 10.3.
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.

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

defun remove-method (generic-function-method)
  ; Remove method.

(defun find-method (generic-function qualifiers specializers [error])
  ; Return suitable method, or signal error.

(defun compute-applicable-methods (generic-function args)
  ; List of methods suitable for args, most specific first.

(defun call-next-method arg current args)
  ; From within a method, call next method with args; return
  ; its values.

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

(defun invalid-method-error (method control arg*)
  ; Signal error on applicable method with invalid qualifiers,
  ; or on method combination. For control and arg* see
  ; format, page 36.

(defun no-next-method (generic-function method arg*)
  ; Called on invocation of call-next-method when there is no
  ; next method. Default method signals error. Not to be called
  ; by user.

(defun function-keywords (method)
  ; Return list of keyword parameters of method and # if other
  ; keys are allowed.

(defun method-qualifiers (method)
  ; List of qualifiers of method.
```

### 10.3 Method Combination Types

#### standard

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

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

(defun define-method-combination (c-type
  ; :documentation string
  (:identity-with-one-argument bool)
  (:operator operator)

```
Short Form. Define new method-combination c-type. In a
generic function using c-type, evaluate most specific
around method supplying the values of the generic func-
tion. 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, re-
turn from the calling call-next-method or from the
generic function, respectively, the values of (operator
(primary-method gen-arg*)*), gen-arg* being the arguments
of the generic function. The primary-methods are ordered
- most-specific-first
- most-specific-last
- most-specific-first
(defgeneric). Using c-type as the qualifier in defmethod
makes the method primary.

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-\lambda* bound to c-arg* (cf. defgeneric), 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 meth-
ods. An applicable method becomes a member of the left-
most group whose predicate or qualifiers match. Methods
can be called via call-method. Lambda lists (ord-\lambda*) and
(method-combination-\lambda*) according to ord-\lambda on page 17,
the latter enhanced by an optional &whole argument.

11 Conditions and Errors

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
:set constructor; it is readable via (reader i) or (accessor i),
and writeable via (writer value i) or (setf accessor i) values.
With allocation class, slot is shared by all conditions of type
foo. A condition is reported by string or by report-function
of arguments condition and stream.
(make-condition condition-type ([\range]\initary-name value))
> Return new instance of condition-type.

(signal form)
(warn form)
(error form)
> 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 ([\range]\initary-name value))
  (control arg))
> Unless handled, signal as correctable error condition or a new instance of condition-type or, with \format control and args (see page 36), 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 errors, NIL and the condition.

:invoke-debugger condition)
> Invoke debugger with condition.

(assert test [\place]
  (condition continue-arg)
  (condition-type ([\range]\initary-name value))
  (control arg))
> If test, which may depend on places, returns NIL, signal as correctable error condition or a new instance of condition-type or, with \format control and args (see page 36), error. When using the debugger’s continue option, places can be altered before re-evaluation of test. Return NIL.

(handler-case foo (\var) (declare decl*) condition-form)*
  [\no-error (ord-\lambda*) (declare decl*) form*)]
> 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*s to values 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*)
> Return values of forms after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.

(with-simple-restart (restart form) control arg*) form*)
> 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 arg-function]
  [\report-function string MINIMAL]
  [\test-function])
  (declare decl*) restart-form*)
> 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*. 

Common Lisp Quick Reference

28
\begin{align*}
\textsf{restart-bind} & \left( \begin{array}{l}
\textsf{restart} \\
\textsf{NIL}
\end{array} \right) \textsf{restart-function} \\
\textsf{restart-function} & \left( \begin{array}{l}
\textsf{interactive-function} \\
\textsf{arg-function} \\
\textsf{report-function} \\
\textsf{test-function} \\
\end{array} \right)^* \textsf{form}\newcommand{\textsf{form}}*\times
\end{align*}

\begin{itemize}
\item Return values of forms evaluated with dynamically established \textsf{restart}s whose \textsf{restart-function}s should perform a non-local transfer of control. A \textsf{restart} is visible under condition if \textsf{(test-function condition)} returns T. If presented in the debugger, \textsf{restart}s are described by \textsf{(invoke-restart \textsf{restart} arg\newcommand{\textsf{arg}}\times)}, where \textsf{args} must be suitable for the corresponding \textsf{restart-function}, or by \textsf{(invoke-restart-interactively \textsf{restart})} where a list of the respective \textsf{args} is supplied by \textsf{arg-function}.
\end{itemize}

\begin{itemize}
\item \textsf{(invoke-restart \textsf{restart} arg\newcommand{\textsf{arg}}\times)}
\item \textsf{(invoke-restart-interactively \textsf{restart})}
\end{itemize}

\begin{align*}
\textsf{return} & \text{values of forms evaluated with dynamically established \textsf{restart}s whose \textsf{restart-function}s should perform a non-local transfer of control. A \textsf{restart} is visible under condition if \textsf{(test-function condition)} returns T. If presented in the debugger, \textsf{restart}s are described by \textsf{(invoke-restart \textsf{restart} arg\newcommand{\textsf{arg}}\times)}, where \textsf{args} must be suitable for the corresponding \textsf{restart-function}, or by \textsf{(invoke-restart-interactively \textsf{restart})} where a list of the respective \textsf{args} is supplied by \textsf{arg-function}.
\end{align*}

\begin{align*}
\textsf{(find-restart \textsf{name}} \textsf{(condition)}) & \text{Return innermost \textsf{restart} \textsf{name}, or a list of all \textsf{restarts}, respectively, out of those either associated with \textsf{condition} or un-associated at all; or, without \textsf{condition}, out of all \textsf{restarts}. Return \textsf{NIL} if search is unsuccessful.}
\end{align*}

\begin{itemize}
\item \textsf{with-condition-restarts condition \textsf{restarts} \textsf{form}\newcommand{\textsf{form}}\times}
\item \textsf{arithmetic-error-operation condition}
\item \textsf{arithmetic-error-operands condition}
\item \textsf{cell-error-name condition}
\item \textsf{unbound-slot-instance condition}
\item \textsf{print-not-readable-object condition}
\item \textsf{package-error-package condition}
\item \textsf{file-error-pathname condition}
\item \textsf{stream-error-stream condition}
\item \textsf{type-error-datum condition}
\item \textsf{type-error-expected-type condition}
\item \textsf{simple-condition-format-control condition}
\item \textsf{simple-condition-format-arguments condition}
\end{itemize}

\begin{align*}
\textsf{break-on-signals} & \text{Condition type debugger is to be invoked on.}
\end{align*}
12 Types and Classes

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

- `(typep foo type [environment])`  \(\triangleright\) If foo is of type.
- `(subtypep type-a type-b [environment])`  \(\triangleright\) Return `T` if type-a is a recognizable subtype of type-b, and `NIL` if the relationship could not be determined.
- `(the type form)`  \(\triangleright\) Declare values of form to be of type.
- `(coerce object type)`  \(\triangleright\) Coerce object into type.
- `(stream-element-type stream)`  \(\triangleright\) Type of stream objects.
- `(array-element-type array)`  \(\triangleright\) Element type array can hold.
- `(upgrade-array-element-type type [environment])`  \(\triangleright\) Element type of most specialized array capable of holding elements of type.
- `(eql foo)`  \(\triangleright\) As a type argument (cf. Figure 2): no restriction.

- `(debugger-hook)`  \(\triangleright\) Function of condition and function itself. Called before debugger.
Figure 2: Precedence Order of System Classes ( ), Classes ( ),
Types ( ), and Condition Types ( ).
Every type is also a supertype of NIL, the empty type.
13 Input/Output

13.1 Predicates

- (f streamp foo) ⇒ T if foo is of indicated type.
- (f pathnamep foo) ⇒ T if foo matches wildcard.
- (f readtablep foo) ⇒ T if foo is a readtable.
- (f input-stream-p stream) ⇒ T if stream is for input.
- (f output-stream-p stream) ⇒ T if stream is for output.
- (f interactive-stream-p stream) ⇒ T if stream is interactive.
- (f open-stream-p stream) ⇒ T if stream is open.
- (f pathname-match-p path wildcard) ⇒ T if path matches wildcard.
- (f wild-pathname-p path) ⇒ T if indicated component in path is wildcard. (NIL indicates any component.)

13.2 Reader

- (f y-or-n-p [control arg]∗) ⇒ Ask user a question and return T or NIL depending on their answer. See page 36 for control and args.
- (f yes-or-no-p [control arg]∗) ⇒ Evaluate forms with standard behaviour of reader and printer. Return values of forms.
- (f read-preserving-whitespace stream [eof-error T [eof-val NIL [recursive NIL]]]) ⇒ Read printed representation of object.
- (f read-from-string string [eof-error T [eof-val NIL [start start [end end [preserve-whitespace bool]]]]]) ⇒ Return object read from string and zero-indexed position of next character.
- (f read-delimited-list char stream) ⇒ Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.
- (f read-byte stream [eof-error [eof-val NIL [recursive NIL]]]) ⇒ Read next byte from binary stream.
- (f read-line stream [eof-error [eof-val NIL [recursive NIL]]]) ⇒ Return a line of text from stream and T if line has been ended by end of file.
(read-sequence sequence stream [start end])
> 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 readtable)

(copy-readtable [from-readtable to-readtable NIL])
> Return copy of from-readtable.

(set-syntax-from-char to-char from-char [to-readtable v *readtable *])
> Copy syntax of from-char to to-readtable. Return T.

(set-macro-character char function [non-term-p NIL [rt v *readtable *]])
> Make char a macro character associated with function of stream and char. Return T.

(get-macro-character char [rt v *readtable *])
> Reader macro function associated with char, and T if char is a non-terminating macro character.

(make-dispatch-macro-character char sub-char function [rt v *readtable *])
> 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 [rt v *readtable *])
> Dispatch function associated with char followed by sub-char.

13.3 Character Syntax

#| multi-line-comment* #| one-line-comment* 
> Comments. There are stylistic conventions:

;;; title
> Short title for a block of code.

;;; intro
> Description before a block of code.

;; state
> State of program or of following code.

; explanation
; continuation
> Regarding line on which it appears.

(food* , bar) 
> List of foods with the terminating cdr bar.

" 
> Begin and end of a string.

'foo
> (quote foo); foo unevaluated.

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

#\c
> (\character *e*), the character c.

#Bn: #0n; #n; #Xn; #Rn
> Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.
Common Lisp Quick Reference

\[ n/d \quad \triangleright \quad \text{the ratio } \frac{n}{d}. \]

\( \{m,n\} \left[ \left\{ \text{short-float, single-float, double-float, long-float} \right\} x \right\} \quad \triangleright \quad m.n \cdot 10^x \) as \#read-default-float-format.

\#C(a,b) \quad \triangleright \quad (\text{complex } a + bi).

\#'(foo) \quad \triangleright \quad (\text{function } foo); \text{ the function named } foo.

\#nAsquence \quad \triangleright \quad n\text{-dimensional array.}

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

\#\( \{\text{foo}\} \text{ }^*\) \quad \triangleright \quad \text{Bit vector of some (or } n\text{) }\text{bs filled with last }\text{b}\text{ if necessary.}

\#S(\text{type } \{\text{slot value}\}) \quad \triangleright \quad \text{structure of }\text{type}.

\#P\text{string} \quad \triangleright \quad \text{A pathname.}

#:\text{foo} \quad \triangleright \quad \text{Uninterned symbol }\text{foo}.

#:\text{form} \quad \triangleright \quad \text{read-time value of }\text{form}.

*:\text{read-eval-\text{foo}} \quad \triangleright \quad \text{If }\text{NIL}, \text{a reader-error is signalled at }\#;\text{.}

#:\text{integer\=\text{foo}} \quad \triangleright \quad \text{Give }\text{foo}\text{ the label }\text{integer}.

#:\text{integer\#} \quad \triangleright \quad \text{Object labelled }\text{integer}.

#:< \quad \triangleright \quad \text{Have the reader signal reader-error.}

#:+\text{feature when-\text{feature}}

#:–\text{feature unless-\text{feature}}

\#\{\text{features}\} \quad \triangleright \quad \text{List of symbols denoting implementation-dependent features.}

\|c\|:c \quad \triangleright \quad \text{Treat arbitrary character(s) }\text{c}\text{ as alphabetic preserving case.}

13.4 Printer

\{\text{print1} \quad \text{print} \quad \text{pprint} \quad \text{princ}\} \quad \text{foo [stream standard-output]}

\triangleright \quad \text{Print }\text{foo}\text{ to }\text{stream}\text{ readably, readably between a newline and a space, readably after a newline, or human-readably without any extra characters, respectively. print1, print and princ return }\text{foo}.\}

\{\text{print1-to-string }\text{foo} \quad \text{print1-to-string }\text{foo}\}

\triangleright \quad \text{Print }\text{foo}\text{ to }\text{string}\text{ readably or human-readably, respectively.}\}

\{\text{print-object }\text{object }\text{stream}\}

\triangleright \quad \text{Print }\text{object}\text{ to }\text{stream}. \text{Called by the Lisp printer.}\}

\{\text{print-unreadable-object }\text{foo }\text{stream} \quad \{\text{type foo identity foo }\text{ form}\}\}

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

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

\triangleright \quad \text{Output a newline to }\text{stream}. \text{Return }\text{NIL}.\}

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

\triangleright \quad \text{Output a newline to }\text{stream}\text{ and return }\text{T}\text{ unless }\text{stream}\text{ is already at the start of a line.}\}
\[\text{\texttt{\textbackslash{}write-char\ char\ [\textbackslash{}stream\ {\texttt{\textbackslash{}standard-output}}]}}\]  
\> Output \texttt{char} to \texttt{stream}.

\[\text{\texttt{\textbackslash{}write-string\ string\ [\textbackslash{}stream\ {\texttt{\textbackslash{}standard-output}}\ {\texttt{[\texttt{\textbackslash{}start\ start}]}\ {\texttt{\textbackslash{}end\ end}}}]}\]  
\> Write \texttt{string} to \texttt{stream} without/with a trailing newline.

\[\text{\texttt{\textbackslash{}write-byte\ byte\ stream}}\]  
\> Write \texttt{byte} to binary stream.

\[\text{\texttt{\textbackslash{}write-sequence\ sequence\ stream\ {\texttt{[\texttt{\textbackslash{}start\ start}]}\ {\texttt{\textbackslash{}end\ end}}}]}\]  
\> Write elements of \texttt{sequence} to binary or character stream.

\[\text{\texttt{\textbackslash{}write\ [\textbackslash{}write-to-string\ foo\ stream\ {\texttt{\textbackslash{}standard-output}}]}}\]  
\> Print \texttt{foo} to \texttt{stream} and return \texttt{foo}, or print \texttt{foo} into string, respectively, after dynamically setting printer variables corresponding to keyword parameters (\texttt{\textbackslash{}print-bar\ becoming\ \textbackslash{}bar\}).

\[\text{\texttt{(\textbackslash{}stream\ keyword\ with\ \textbackslash{}write\ only.)}}\]

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

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

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

\[\text{\texttt{\textbackslash{}pprint-dispatch\ \texttt{dispatch-table\ \texttt{\{\texttt{\textbackslash{}pprint-logical-block\ [\texttt{stream\ list\ {\texttt{\{\texttt{\textbackslash{}prefix\ string\}\ {\texttt{\textbackslash{}per-line-prefix\ string\]}\}\ {\texttt{\textbackslash{}suffix\ string\}}\}}]}\}}}\)

\[\text{\texttt{(\texttt{declare\ decl\})\ *\ form\}}\]  
\> Evaluate \texttt{forms}, which should print \texttt{list}, with \texttt{stream} locally bound to a pretty printing stream which outputs to the original \texttt{stream}. If \texttt{list} is in fact not a list, it is printed by \texttt{\textbackslash{}write\}. Return \texttt{NIL}

\[\text{\texttt{\textbackslash{}pprint-pop\ \texttt{\textbackslash{}\}}\]  
\> Take next element off \texttt{list}. If there is no remaining tail of \texttt{list}, or \texttt{\textbackslash{}pprint-length\} or \texttt{\textbackslash{}pprint-circle\} indicate printing should end, send element together with an appropriate indicator to \texttt{stream}.

\[\text{\texttt{\textbackslash{}pprint-tab\ \texttt{(line\ line-relative\ section\ section-relative\ \texttt{\textbackslash{}\}}\]  
\> Move cursor forward to column number \(c + ki\), \(k \geq 0\) being as small as possible.

\[\text{\texttt{\textbackslash{}pprint-indent\ \texttt{(block\ current\ \texttt{\textbackslash{}\}}\]  
\> Specify indentation for innermost logical block relative to leftmost position/to current position. Return \texttt{NIL}

\[\text{\texttt{\textbackslash{}pprint-exit-if-list\-exhausted\ \texttt{\textbackslash{}\}}\]  
\> If \texttt{list} is empty, terminate logical block. Return \texttt{NIL} otherwise.
Common Lisp Quick Reference

(print-newline [linear | fill | miser | mandatory] [stream <standard-output>])
 ▶ Print a conditional newline if stream is a pretty printing stream. Return NIL.

(print-array* [T] If T, print arrays readably.
(print-base* [M] Radix for printing rationals, from 2 to 36.
(print-case* [M] Print symbol names all uppercase (:upcase), all lowercase (:downcase), capitalized (:capitalize).
(print-circle* [NIL] If T, avoid indefinite recursion while printing circular structure.
(print-escape* [T] If NIL, do not print escape characters and package prefixes.
(print-gensym* [T] If T, print #: before uninterned symbols.
(print-length* [NIL] If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.
(print-miser-width* [T] If integer and greater than the width available for printing a substructure, switch to the more compact miser style.
(print-pretty* [T] If T, print prettily.
(print-radix* [NIL] If T, print rationals with a radix indicator.
(print-readably* [T] If T, print readably or signal error print-not-readable.
(print-right-margin* [NIL] Right margin width in ems while pretty-printing.
(set-pprint-dispatch type function [priority] [table <print-pprint-dispatch>]) ▶ Install entry comprising function of arguments stream and object to print; and priority as type into table. If function is NIL, remove type from table. Return NIL.
(pprint-dispatch foo [table <print-pprint-dispatch>]) ▶ Return highest priority function associated with type of foo and T if there was a matching type specifier in table.
(copy-pprint-dispatch [table <print-pprint-dispatch>]) ▶ Return copy of table or, if table is NIL, initial value of *print-pprint-dispatch*.

*print-pprint-dispatch* ▶ Current pretty print dispatch table.

13.5 Format

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

(format {T}<NIL>out-string out-stream control arg*) ▶ Output string control which may contain ← directives possibly taking some args. Alternatively, control can be a function returned by(formatter 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. If first argument is NIL, return formatted output.
Common Lisp Quick Reference

- \([\text{min-col}][\text{col-inr}][\text{min-pad}][\text{pad-char}]\)
  ➤ Aesthetic/Standard. Print argument of any type for consumption by humans/by the reader, respectively. With \(;\), print \(\text{NIL}\) as \(\text{NIL}\) rather than \(\text{nil}\); with \(\Theta\), add \text{pad-char}s on the left rather than on the right.

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

- \([\text{roman}][\text{width}][\text{pad-char}][\text{comma-char}]\)
  ➤ Roman. Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

- \([\text{width}][\text{pad-char}][\text{comma-char}]\)
  ➤ Decimal/Binary/Octal/Hexadecimal. Print integer argument as number. With \(;\), group digits comma-interval each; with \(\Theta\), always prepend a sign.

- \([\text{width}][\text{dec-digits}][\text{shift}]\)
  ➤ Fixed-Format Floating-Point. With \(\Theta\), always prepend a sign.

- \([\text{width}][\text{dec-digits}][\text{exp-digits}][\text{scale-factor}]\)
  ➤ 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 \(-G\), choose either \(+\) or \(-F\). With \(\Theta\), always prepend a sign.

- \([\text{width}][\text{int-digits}][\text{comma-interval}]\)
  ➤ Monetary Floating-Point. Print argument as fixed-format floating-point number. With \(;\), put sign before any padding; with \(\Theta\), always prepend a sign.

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

- \([\text{text}][\text{text}]\)
  ➤ Case-Conversion. Convert \text{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.

- \([\text{plur}][\text{op}][\Theta]\)
  ➤ Plural. If argument \text{eql} 1 print nothing, otherwise print \text{a}, do the same for the previous argument; if argument \text{eql} 1 print \(y\), otherwise print \text{ies}; do the same for the previous argument, respectively.

- \([\text{percent}][\text{newline}]\)
  ➤ Newline. Print \(n\) newlines.

- \([\text{fill}][\text{fresh-line}]\)
  ➤ Fresh-Line. Print \(n - 1\) newlines if output stream is at the beginning of a line, or \(n\) newlines otherwise.

- \([\text{conditional-newline}][\text{pprint-newline}][\text{linear}][\text{fill}][\text{miser}][\text{mandatory}][\text{spare}][\text{null-text}][\text{right}]\)
  ➤ Ignored Newline. Ignore newline, or whitespace following newline, or both, respectively.

- \([\text{page}][\text{tildes}]\)
  ➤ Page. Print \(n\) page separators.

- \([\text{null-text}][\text{min-col}][\text{col-inr}][\text{min-pad}][\text{pad-char}]\)
  ➤ Justification. Justify text produced by \text{texts} in a field of at least \text{min-col} columns. With \(;\), right justify; with \(\Theta\), left justify. If this would leave less than \text{spare} characters on the current line, output \text{null-text} first.
Common Lisp Quick Reference

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

  { [ ] [ ] [ ] @ }  
  ⊲ Indent. Set indentation to n relative to leftmost/to current position.

- [ ] [ ] [ ] T  
  ⊲ Tabulate. Move cursor forward to column number c + k, k ≥ 0 being as small as possible. With ;, calculate column numbers relative to the immediately enclosing section. With @, move to column number c0 + c + k where c0 is the current position.

  { [ ] [ ] [ ] * ] [ ] [ ] * [ ] [ ] @ * }  
  ⊲ Go-To. Jump m arguments forward, or backward, or to argument n.

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

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

  [ x , y , z ]  
  ⊲ Conditional Expression. Use the zero-indexed argument (or i-th if given) text as a format control subclause. With ;, use the first text if the argument value is NIL, or the second text if it is T. With @, do nothing for an argument value of NIL. Use the only text and leave the argument to be read again if it is T.

  { [ ] @ }  
  ⊲ Recursive Processing. Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

  [prefix {prefix} ] [ ] @ [package [::] function]/  
  ⊲ Call Function. Call all-uppercase package:function with the arguments stream, format-argument, colon-p, at-sign-p and prefixes for printing format-argument.

  [ ] @ W  
  ⊲ Write. Print argument of any type obeying every printer control variable. With ;, pretty-print. With @, print without limits on length or depth.

  { [ ] [ ] }  
  ⊲ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.
13.6 Streams

(\open path)

- \(\text{open-file-stream-to-path}\)

- \(\text{make-concatenated-stream input-stream}^*\)
- \(\text{make-broadcast-stream output-stream}^*\)
- \(\text{make-two-way-stream input-stream-part output-stream-part}\)
- \(\text{make-echo-stream from-input-stream to-output-stream}\)
- \(\text{make-synonym-stream variable-bound-to-stream}\)

\(\text{Return stream of indicated type.}\)

- \(\text{make-string-input-stream string} \begin{align*} &\begin{cases} \text{start} \quad \text{end} \end{cases} \end{align*} \)

\(\text{Return a string-stream supplying the characters from string.}\)

- \(\text{make-string-output-stream} \begin{align*} &\begin{cases} \text{element-type type} \end{cases} \end{align*} \)

\(\text{Return a stream accepting characters (available via } \text{get-output-stream-string}.\)

- \(\text{concatenated-stream-streams concatenated-stream}\)
- \(\text{broadcast-stream-streams broadcast-stream}\)

\(\text{Return list of streams concatenated-stream still has to read from/broadcast-stream is broadcasting to.}\)

- \(\text{two-way-stream-input-stream} \begin{align*} &\begin{cases} \text{two-way-stream} \end{cases} \end{align*} \)
- \(\text{two-way-stream-output-stream}\)
- \(\text{echo-stream-input-stream} \text{ echo-stream}\)
- \(\text{echo-stream-output-stream} \text{ echo-stream}\)

\(\text{Return source stream or sink stream of two-way-stream/echo-stream, respectively.}\)

- \(\text{synonym-stream-symbol synonym-stream}\)

\(\text{Return symbol of synonym-stream.}\)

- \(\text{get-output-stream-string string-stream}\)

\(\text{Clear and return as a string characters on string-stream.}\)

- \(\text{file-position} \begin{align*} &\begin{cases} \text{start} \quad \text{end} \quad \text{position} \end{cases} \end{align*}\)

\(\text{Return position within stream, or set it to position and return } T \text{ on success.}\)

- \(\text{file-string-length} \text{ stream foo}\)

\(\text{Length foo would have in stream.}\)

- \(\text{listen} \begin{align*} &\begin{cases} \text{standard-input} \end{cases} \end{align*}\)

\(T \text{ if there is a character in input stream.}\)

- \(\text{clear-input} \begin{align*} &\begin{cases} \text{standard-input} \end{cases} \end{align*}\)

\(\text{Clear input from stream, return } NIL\)

- \(\text{clear-output} \begin{align*} &\begin{cases} \text{standard-output} \end{cases} \end{align*}\)

\(\text{End output to stream and return } NIL \text{ immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.}\)
Common Lisp Quick Reference

\((\text{close} \ \text{stream} \ \text{[abort bood]}))\)

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

\((\text{with-open-file} \ (\text{stream} \ \text{path} \ \text{open-args}) \ \text{(declare} \ \text{decl}^n*} \ \text{form}^k))\)

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

\((\text{with-open-stream} \ (\text{foo stream}) \ \text{(declare} \ \text{decl}^n*} \ \text{form}^k))\)

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

\((\text{with-input-from-string} \ (\text{foo string}) \ \text{(declare} \ \text{decl}^n*} \ \text{form}^k))\)

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

\((\text{with-output-to-string} \ (\text{foo string}) \ \text{(declare} \ \text{decl}^n*} \ \text{form}^k))\)

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

\((\text{stream-external-format} \ \text{stream})\)

- External file format designator.

\(\text{*terminal-io*}\)

- Bidirectional stream to user terminal.

\(\text{*standard-input*}\)

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

\(\text{*error-output*}\)

\(\text{*debug-io*}\)

\(\text{*query-io*}\)

- Bidirectional streams for debugging and user interaction.

### 13.7 Pathnames and Files

\((\text{make-pathname} \ (\text{[host]} \ \text{NIL} \ \text{unspecific}) \ \text{[device]} \ \text{NIL} \ \text{unspecific}) \ \text{[directory]} \ \text{wild} \ \text{NIL} \ \text{unspecific})\)

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

\(\text{pathname-host}\)

\(\text{pathname-device}\)

\(\text{pathname-directory}\)

\(\text{pathname-name}\)

\(\text{pathname-type}\)

\(\text{pathname-version}\)

- Return pathname component.
\(f\) parse-namestring \(foo\)
- Returns pathname converted from string, pathname, or stream \(foo\); and position where parsing stopped.

\(f\) merge-pathnames \(path-or-stream\)
- Returns pathname made by filling in components missing in \(path-or-stream\) from default-path-or-stream.

\(\ast\) default-pathname-defaults
- Pathname to use if one is needed and none supplied.

\(f\) user-homedir-pathname \([host]\)
- User’s home directory.

\(f\) enough-namestring \(path-or-stream\)
- Returns minimal pathname that sufficiently describes the path of \(path-or-stream\) relative to root-path.

\(f\) file-namestring \(path-or-stream\)
\(f\) directory-namestring \(path-or-stream\)
\(f\) host-namestring \(path-or-stream\)
- Returns string representing full pathname; name, type, and version; directory name; or host name, respectively, of \(path-or-stream\).

\(f\) translate-pathname \(path-or-stream\) wildcard-path-a wildcard-path-b
- Translate the path of \(path-or-stream\) from wildcard-path-a into wildcard-path-b. Return new path.

\(f\) pathname \(path-or-stream\)
- Pathname of \(path-or-stream\).

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

\(f\) translate-logical-pathname \(path-or-stream\)
- Physical pathname corresponding to (possibly logical) pathname of \(path-or-stream\).

\(f\) probe-file \(file\)
\(f\) truename \(file\)
- Canonical name of \(file\). If \(file\) does not exist, return NIL/signal file-error, respectively.

\(f\) file-write-date \(file\)
- Time at which \(file\) was last written.

\(f\) file-author \(file\)
- Return name of file owner.

\(f\) file-length \(stream\)
- Return length of \(stream\).

\(f\) rename-file \(foo\) \(bar\)
- Rename file \(foo\) to \(bar\). Unspecified components of path \(bar\) default to those of \(foo\). Return new pathname, old physical file name, and new physical file name.

\(f\) delete-file \(file\)
- Delete \(file\). Return T.

\(f\) directory \(path\)
- List of pathnames matching \(path\).

\(f\) ensure-directories-exist \(path\) \([verbose \text{ bool}]\)
- Create parts of \(path\) if necessary. Second return value is T if something has been created.
14 Packages and Symbols

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

14.1 Predicates

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

(packagep foo) ⇒ Keyword, evaluates to .bar.

(keywordp foo) ⇒ Possibly unexported symbol of package.

14.2 Packages

:bar|keyword:bar ⇒ Make package foo current.

package::symbol ⇒ Exported symbol of package.

package::symbol ⇒ Possibly unexported symbol of package.

(make-package foo {⟨nicknames nick*⟩IDEOS ⟨use (used-package *)⟩}) ⇒ Create package foo.


(in-package foo) ⇒ 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.

(delete-package foo) ⇒ Delete package. Return T if successful.

(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.

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

(unintern symbol [package]) ⇒ Remove symbol from package, return T on success.
(import (shadowing-import) symbols [package 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 package])
  ▷ Make symbols of package shadow any otherwise accessible, equally named symbols from other packages. Return T.

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

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

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

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

(with-package-iterator (foo packages [internal|external|inherited])
  (declare decl* form*)
  ▷ 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.

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

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

*modules* ▷ List of names of loaded modules.

14.3 Symbols

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

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

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

(gentemp [prefix package package])
  ▷ Intern fresh symbol in package. Deprecated.

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

(symbol-name symbol)

(symbol-package symbol)
  ▷ Name or package, respectively, of symbol.

(symbol-plist symbol)

(symbol-value symbol)

(symbol-function symbol)
  ▷ Property list, value, or function, respectively, of symbol. seffable.
Common Lisp Quick Reference

```lisp
(setf documentation) new-doc
> Get/set documentation string of foo of given type.

T
> Truth; the supertype of every type including t; the super-
class of every class except t; *terminio*.

nil
> Falsity; the empty list; the empty type, subtype of every
type; *standard-input*; *standard-output*; the global envi-
ronment.

14.4 Standard Packages

common-lisp
> Exports the defined names of Common Lisp except for
those in the keyword package.

common-lisp-user
> Current package after startup; uses package common-lisp.

keyword
> Contains symbols which are defined to be of type keyword.

15 Compiler

15.1 Predicates

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

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

15.2 Compilation

(compile name)
> Return compiled function or replace name’s function def-
inition with the compiled function. Return T in case of
warnings or errors, and NIL in case of warnings or errors ex-
cluding style-warnings.

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

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

(compile-file path)
> Input file used by compile-file/ by load.

(load)
> Defaults used by compile-file/ by load.
```
(eval-when (compile-toplevel load execute) form)

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

(locally (declare decl*) form)

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

(with-compilation-unit (':override bool) form)

▷ Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of forms.

(load-time-value form [read-only])

▷ Evaluate form at compile time and treat its value as literal at run time.

(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-saving-slots foo {[:slot-names slots] :environment environment})

▷ Return a creation form and an initialization form which on evaluation construct an object equivalent to foo with slots initialized with the corresponding values from foo.

(f macro-function symbol [environment])

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

(f eval arg)

▷ Return values of value of arg evaluated in global environment.

15.3 REPL and Debugging

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

← Form currently being evaluated by the REPL.

(apropos string [package])

▷ Print interned symbols containing string.

(apropos-list string [package])

▷ List of interned symbols containing string.

(dribble [path])

▷ Save a record of interactive session to file at path. Without path, close that file.

(ed [file-or-function])

▷ Invoke editor if possible.

{macroexpand-1} form [environment]

▷ Return macro expansion, once or entirely, respectively, of form and if form was a macro form. Return form and NIL otherwise.

*:macroexpand-hook*

▷ Function of arguments expansion function, macro form, and environment called by :macroexpand-1 to generate macro expansions.
(mtrace {function {setf function} } *)
▷ Cause functions to be traced. With no arguments, return
list of traced functions.

(muntrace {function {setf function} } *)
▷ Stop functions, or each currently traced function, from be-
ing traced.

•trace-output
▷ Output stream mtrace and mtime send their output to.

•step form
▷ Step through evaluation of form. Return values of form.

•break [control arg*]]
▷ Jump directly into debugger; return NIL. See page 36,
•format, for control and args.

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

•inspect foo
▷ Interactively give information about foo.

•describe foo stream
▷ Send information about foo to stream. Called by •describe.

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

•room [NIL:default:T]:default]
▷ Print information about internal storage management to
•standard-output*.

15.4 Declarations

•proclaim decl

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

(declare decl*)
▷ Inside certain forms, locally make declarations decl*. decl
can be: dynamic-extent, type, ftype, ignorable, ignore, inline,
otinline, 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]*)
(ftype function*)
▷ Declare variables or functions to be of type.

([ignore] {var} {function function}*)
▷ Suppress warnings about used/unused bindings.

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

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

(special var*)
▷ Declare vars to be dynamic.
16 External Environment

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

(get-internal-run-time)  ▶ Number of clock ticks per second.

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

(encode-universal-time sec min hour date month year [zone])

(get-decoded-time)  ▶ Return second, minute, hour, date, month, year, day, daylight-p, and zone.

(short-site-name)

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

(lisp-implementation)  ▶ Name or version of implementation, operating system, or hardware, respectively.

/software

/machine  ▶ Computer name.