Quick Reference

cl

Common

lisp

Bert Burgemeister
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 Seq. Predicates . . 12
  6.2 Seq. Functions . . 12

7 Hash Tables 14

8 Structures 15

9 Control Structure 15
  9.1 Predicates . . . . 15
  9.2 Variables . . . . 16
  9.3 Functions . . . . 17
  9.4 Macros . . . . . . 18

10 CLOS 24
  10.1 Classes . . . . . 24
  10.2 Generic Functions . 25
  10.3 Method Combination Types . . . 26

11 Conditions and Errors 27

12 Types and Classes 30

13 Input/Output 32

14 Packages and Symbols 42
  14.1 Predicates . . . . 42
  14.2 Packages . . . . 42
  14.3 Symbols . . . . 43

15 Compiler 44
  15.1 Predicates . . . . 44
  15.2 Compilation . . . . 44
  15.3 REPL & Debug . . . . 45
  15.4 Declarations . . . . 46

16 External Environment 47

Typographic Conventions

name; name; name; name; name; name; name; name; name;
▷ 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} ▷ Anything from none to each of foo, bar, or 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 n-th return value.

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

1.1 Predicates

(\(=\) number\(^+\)) \(\implies\) T if all numbers, or none, respectively, are equal in value.

(\(\geq\) number\(^+\)) \(\implies\) T if numbers are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

\(\text{(minusp a)}\) \(\implies\) T if \(a < 0\), \(a = 0\), or \(a > 0\), respectively.

\(\text{(evenp int)}\) \(\implies\) T if \(\text{int}\) is even or odd, respectively.

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

1.2 Numeric Functions

\(\sum a\) or \(\prod a\), respectively.

\(a - \sum b\) or \(a/\prod b\), respectively. Without any \(b\), return \(-a\) or \(1/a\), respectively.

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

\(\text{increment \ or \ decrement \ the \ value \ of \ } place\ \text{by} \ delta. \ \text{Return} \ \text{new \ value.}\)

\(a^p\) or \(b^p\), respectively.

\(\log a\) or, without \(b\), \(\ln a\).

\(\sqrt{\text{in complex numbers/natural numbers.}}\)

\(\text{least common multiple or greatest common denominator, respectively, of integers}. \ \text{(gcd) returns 0.}\)

\(\text{long-float approximation of } \pi, \ \text{Ludolph’s number.}\)

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

\(\arcsin a\) or \(\arccos a\), respectively, in radians.

\(\text{arctan a in radians.}\)

\(\text{sinh a} \ \text{cosh a} \ \text{or tanh a}, \text{respectively.}\)
Common Lisp Quick Reference

\[(\text{asin} \ a) \quad \triangleright \quad \text{asinh} \ a, \ \text{acosh} \ a, \text{or} \ \text{atanh} \ a, \text{respectively.}\]

\[(\text{atan} \ a) \quad \triangleright \quad \text{Return} \ \sqrt{a^2 - 1} = \cos a + i \sin a.\]

\[(\text{conjugate} \ a) \quad \triangleright \quad \text{Return complex conjugate of} \ a.\]

\[(\text{max} \ \text{num} \ ) \quad \triangleright \quad \text{Greatest or least, respectively, of} \ \text{num}s.\]

\[(\text{min} \ \text{num} \ ) \quad \triangleright \quad \text{Return} \ e^{i a} = \cos a + i \sin a.\]

\[(\text{cis} \ a) \quad \triangleright \quad \text{Return complex number} \ e^{i a}.\]

\[(\text{conjugate} \ a) \quad \triangleright \quad \text{Return complex conjugate of} \ a.\]

\[(\text{round} \ \text{floor} \ \text{ceiling} \ \text{truncate}) \quad \triangleright \quad \text{Return} \ a \text{ as integer or float, respectively,} \ n/d \text{ rounded, or rounded towards} \ -\infty, +\infty, \text{or} \ 0, \text{respectively; and remainder.}\]

\[(\text{mod} \ \text{rem}) \quad \triangleright \quad \text{Same as} \ \text{floor} \text{ or} \ \text{truncate}, \text{respectively, but return remainder only.}\]

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

\[(\text{make-random-state}) \quad \triangleright \quad \text{Copy of random-state object} \ \text{or of the current random state; or a randomly initialized fresh random state.}\]

\[(\text{v \ random-state}) \quad \triangleright \quad \text{Current random state.}\]

\[(\text{realpart} \ \text{imagpart}) \quad \triangleright \quad \text{Real part or imaginary part, respectively, of} \ \text{number.}\]

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

\[(\text{phase} \ \text{num}) \quad \triangleright \quad \text{Angle of} \ \text{num}'s \text{ polar representation.}\]

\[(\text{abs} \ \text{n}) \quad \triangleright \quad \text{Return} \ |n|.\]

\[(\text{rational} \ \text{real}) \quad \triangleright \quad \text{Convert} \ \text{real to rational. Assume complete/limited accuracy for} \ \text{real.}\]

\[(\text{float} \ \text{real} \ \text{prototype}) \quad \triangleright \quad \text{Convert real into float with type of} \ \text{prototype.}\]

### 1.3 Logic Functions

Negative integers are used in two’s complement representation.

\[(\text{boole operation} \ \text{int-a} \ \text{int-b}) \quad \triangleright \quad \text{Return value of bitwise logical operation. operations are}\]

- \text{boole-1} \quad \triangleright \quad \text{int-a.}\n- \text{boole-2} \quad \triangleright \quad \text{int-b.}\n- \text{boole-c1} \quad \triangleright \quad \lfloor\text{int-a}\rfloor\n- \text{boole-c2} \quad \triangleright \quad \lceil\text{int-b}\rceil\n- \text{boole-set} \quad \triangleright \quad \text{All bits set.}\n- \text{boole-clr} \quad \triangleright \quad \text{All bits zero.}\]
1.4 Integer Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  ▶ Available numbers closest to \(-0\) or \(+0\), respectively.

- **most-negative**
  - **most-positive**

  ▶ Available numbers closest to \(-\infty\) or \(+\infty\), respectively.

- **epsilon**
  - **negative-epsilon**

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

- **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**

  ▶ Available numbers closest to \(-\infty\) or \(+\infty\), respectively.

- **fixnum**

  ▶ Available numbers closest to \(-\infty\) or \(+\infty\), respectively.

2 Characters

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

- **characterp**
  - **standard-char-p**

    ▶ T if argument is of indicated type.

- **graphic-char-p**
  - **alpha-char-p**
  - **alnumterp**

    ▶ T if character is visible, alphabetic, or alphanumeric, respectively.

- **upper-case-p**
  - **lower-case-p**
  - **both-case-p**

    ▶ Return T if character is uppercase, lowercase, or able to be in another case, respectively.

- **digit-char-p**

    ▶ Return its weight if character is a digit, or NIL otherwise.

- **char=**
  - **char/=**

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

- **char-between**
  - **char-not-between**

    ▶ Return T if all characters, or none, respectively, are equal ignoring case.

- **char>**
  - **char>=''
  - **char<=''
  - **char<**

    ▶ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.
Common Lisp Quick Reference

3 Strings

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

(defun simple-string-p foo)  \(\Rightarrow T\) if \(foo\) is of indicated type.

(defun string-equal foo bar)  \(\Rightarrow T\) if subsequences of \(foo\) and \(bar\) are equal. Obey/ignore, respectively, case.

(defun string-upcase \(\{\) start \(\) \}) \(\Rightarrow\) Convert \(x\) \(\{\text{symbol, string, or character}\}\) into a string, a string with capitalized words, an all-upper case string, or an all-lower case string, respectively.

(defun string-downcase \(\{\) start \(\) \}) \(\Rightarrow\) Convert \(\text{string}\) into a string with capitalized words, an all-upper case string, or an all-lower case string, respectively.

(defun string-trim \(\{\) char-bag \(\) \}) \(\Rightarrow\) Return \(\text{string}\) with all characters in sequence \(\text{char-bag}\) removed from both ends, from the beginning, or from the end, respectively.

(defun digit-char \(\{\) radix \(\) \}) \(\Rightarrow\) Character representing digit \(i\).
(character string i)  
> Return zero-indexed i-th character of string ignoring/obeying, respectively, fill pointer. setfable.

(parse-integer string i)  
> Return integer parsed from string and index of parse end.

### 4 Conses

#### 4.1 Predicates

(consf foo)  
> Return T if foo is of indicated type.

(endp list)  
> Return T if list/foo is NIL.

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

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

#### 4.2 Lists

(cons foo bar)  
> Return new cons (foo . bar).

(list foo)  
> Return list of foo.

(list* foo+)  
> Return list of foo with last foo becoming cdr of last cons. Return foo if only one foo given.

(make-list num [initial-element foo NIL])  
> New list with num elements set to foo.

(list-length list)  
> Length of list; NIL for circular list.

(car list)  
> Car of list or NIL if list is NIL. setfable.

(cdr list)  
> Cdr of list or NIL if list is NIL. setfable.

(cdrcdr n list)  
> Return tail of list after calling cdr n times.

(first list)  
> Return nth element of list if any, or NIL otherwise. setfable.

(last list num)  
> Return list of last num conses of list.

(cadr list)  
> With X being one to four as and ds representing cars and cdrs, e.g. (cadr bar) is equivalent to (car (cadr bar)). setfable.
\begin{itemize}
\item \texttt{butlast list} \quad \texttt{num}\[\text{list}\] \quad \triangleright \quad \text{list excluding last \texttt{num} conses.}
\item \texttt{butlast list} \quad \texttt{num}\[\text{list}\] \quad \triangleright \quad \text{list excluding last \texttt{num} conses.}
\item \texttt{diff list foo} \quad \triangleright \quad \text{If \texttt{foo} is a tail of \texttt{list}, return preceding part of \texttt{list}. Otherwise return \texttt{list}.}
\item \texttt{rplaca f rplacd} \quad \texttt{cons object} \quad \triangleright \quad \text{Replace \texttt{car}, or \texttt{cdr}, respectively, of \texttt{cons} with \texttt{object}.}
\item \texttt{rplaca f rplacd} \quad \texttt{list} \quad \triangleright \quad \text{Replace \texttt{car}, or \texttt{cdr}, respectively, of \texttt{cons} with \texttt{object}.}
\item \texttt{fplaca f rplacd} \quad \texttt{list} \quad \triangleright \quad \text{Replace \texttt{car}, or \texttt{cdr}, respectively, of \texttt{cons} with \texttt{object}.}
\item \texttt{adjoin foo list} \quad \triangleright \quad \text{Return \texttt{list} if \texttt{foo} is already member of \texttt{list}. If not, return \texttt{(cons foo list)}.}
\item \texttt{pop place} \quad \triangleright \quad \text{Set \texttt{place} to \texttt{(cdr place)}, return \texttt{(car place)}.}
\item \texttt{push foo place} \quad \triangleright \quad \text{Set \texttt{place} to \texttt{(cons foo place)}.}
\item \texttt{pushnew foo place} \quad \triangleright \quad \text{Set \texttt{place} to \texttt{(adjoin foo place)}.}
\item \texttt{append proper-list * foo NIL} \quad \triangleright \quad \text{Return concatenated list or, with only one argument, \texttt{foo}, \texttt{foo} can be of any type.}
\item \texttt{revappend list foo} \quad \triangleright \quad \text{Return concatenated list after reversing order in \texttt{list}.}
\item \texttt{copy-list list} \quad \triangleright \quad \text{Return copy of \texttt{list} with shared elements.}
\end{itemize}

\section{Association Lists}
\begin{itemize}
\item \texttt{pairlis keys values [alias list]} \quad \triangleright \quad \text{Prepend to \texttt{alist} an association list made from lists \texttt{keys} and \texttt{values}.}
\item \texttt{acons key value alist} \quad \triangleright \quad \text{Return \texttt{alist} with a \texttt{(key . value)} pair added.}
\item \texttt{assoc foo alist} \quad \triangleright \quad \text{First \texttt{cons} whose \texttt{car}, or \texttt{cdr}, respectively, satisfies \texttt{test}.}
\item \texttt{copy-alist alist} \quad \triangleright \quad \text{Return copy of \texttt{alist}.}
\end{itemize}
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.

{subst-if-not new test tree} {key function}
\> Make copy of tree with each subtree or leaf satisfying test replaced by new.

{sublis association-list tree} {test function eql}
\> Make copy of tree with each subtree or leaf matching a key in association-list replaced by that key’s value.

4.5 Sets

\begin{align*}
\text{intersection} & \quad a \cap b \\
\text{set-difference} & \quad a \setminus b \\
\text{union} & \quad a \cup b \\
\text{set-exclusive-or} & \quad a \triangle b
\end{align*}
\> Return \(a \cap b, a \setminus b, a \cup b, \text{or } a \triangle b\), respectively, of lists \(a\) and \(b\).

5 Arrays

5.1 Predicates

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

5.2 Array Functions

\begin{align*}
\text{make-array} & \quad \text{dimension-sizes} & \quad \text{adjustable boolean} \\
\text{adjust-array} & \quad \text{array dimension-sizes} & \quad \text{element-type type} \\
\text{aref} & \quad \text{array \{subscripts\}} & \quad \text{row-major-aref array i}
\end{align*}
\> Return fresh, or readjust, respectively, vector or array.

\begin{align*}
\text{aref} \quad \text{array \{subscripts\}} & \quad \text{Return array element pointed to by subscripts. settable.} \\
\text{row-major-aref array i} & \quad \text{Return i-th element of array in row-major order. settable.}
\end{align*}
(array-row-major-index array [subscripts])
  ▷ Index in row-major order of the element denoted by subscripts.

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

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

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

(array-rank array)
  ▷ Number of dimensions of array.

(array-displacement array)
  ▷ Target array and offset.

(bit array [subscripts])
  (sbite simple-bit-array [subscripts])
  ▷ Return element of bit-array or of simple-bit-array. setfable.

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

(bit-eqv bit-and bit-andc1 bit-andc2 bit-nand bit-ior bit-orc1 bit-orc2 bit-xor bit-nor)
  ▷ Return result of bitwise logical operations (cf. operations of f:boole, page 4) on bit-array-a and bit-array-b. If result-bit-array is T, put result in bit-array-a; if it is NIL, make a new array for result.

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

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

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

5.3 Vector Functions

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

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

(svref vector i)
  ▷ Element i of simple vector. setfable.

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

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

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

(fill-pointer vector)
  ▷ Fill pointer of vector. setfable.
6 Sequences

6.1 Sequence Predicates

\{every \notevery\} test sequence+  
\hfill Return NIL or T, respectively, as soon as test on any set of corresponding elements of sequences returns NIL.

\{some \notany\} test sequence+  
\hfill Return value of test or NIL, respectively, as soon as test on any set of corresponding elements of sequences returns non-NIL.

\[\text{mismatch sequence-a sequence-b}\]  
\hfill Return position in sequence-a where sequence-a and sequence-b begin to mismatch. Return NIL if they match entirely.

6.2 Sequence Functions

\{\text{make-sequence sequence-type size [initial-element foo]}\}  
\hfill Make sequence of sequence-type with size elements.

\{\text{concatenate type sequence*}\}  
\hfill Return concatenated sequence of type.

\{\text{merge type sequence-a sequence-b test [key function]}\}  
\hfill Return interleaved sequence of type. Merged sequence will be sorted if both sequence-a and sequence-b are sorted.

\{\text{fill sequence foo}\}  
\hfill Return sequence after setting elements between start and end to foo.

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

\{\text{count foo sequence}\}  
\hfill Return number of elements in sequence which match foo.

\{\text{count-if \notcount-if-not\} test sequence}\}  
\hfill Return number of elements in sequence which satisfy test.

\{\text{elt sequence index}\}  
\hfill Return element of sequence pointed to by zero-indexed index. setfable.

\{\text{subseq sequence start [end]}\}  
\hfill Return subsequence of sequence between start and end. setfable.

\{\text{sort \notstable-sort\} sequence test [key function]}\}  
\hfill Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

\{\text{reverse sequence}\}  
\hfill Return sequence in reverse order.

\{\text{reverse sequence}\}  
\hfill Return sequence in reverse order.
Common Lisp Quick Reference

\[
\{\text{find} \mid \text{position}\} \text{ foos sequence} \\
\{\text{find-if} \mid \text{position-if}\} \text{ test sequence} \\
\{\text{search sequence-a} \mid \text{search sequence-b}\} \\
\{\text{remove foos sequence} \mid \text{delete foos sequence}\} \\
\{\text{remove-if} \mid \text{remove-if-not}\} \text{ test sequence} \\
\{\text{remove-duplicates sequence} \mid \text{delete-duplicates sequence}\} \\
\{\text{substitute new old sequence} \mid \text{nsubstitute new old sequence}\} \\
\{\text{substitute-if} \mid \text{substitute-if-not}\} \text{ new test sequence} \\
\{\text{substitute-if} \mid \text{substitute-if-not}\} \text{ new test sequence} \\
\{\text{position relative to the begin of sequence, respectively.}\} \\
\{\text{return first element in sequence which matches foo, or its position relative to the begin of sequence, respectively.}\} \\
\{\text{return first element in sequence which satisfies test, or its position relative to the begin of sequence, respectively.}\} \\
\{\text{search sequence-b for a subsequence matching sequence-a. Return position in sequence-b, or NIL.}\} \\
\{\text{make copy of sequence without elements matching foo.}\} \\
\{\text{make copy of sequence with all (or count) elements satisfying test removed.}\} \\
\{\text{make copy of sequence without duplicates.}\} \\
\{\text{make copy of sequence with all (or count) olds replaced by new.}\} \\
\{\text{make copy of sequence with all (or count) elements satisfying test replaced by new.}\}
\]
(replace sequence-a sequence-b)  
▷ Replace elements of sequence-a with elements of sequence-b.

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

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

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

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

7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see loop, page 21. Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

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

(make-hash-table)  
▷ Make a hash table.

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

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

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

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

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

(with-hash-table-iterator (foo hash-table) (declare (decdash dec+) *form5))  
▷ Return values of forms. In forms, invocations of (foo) return: T if an entry is returned; its key; its value.

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

(hash-table-size hash-table)  

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

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

(defstruct foo
  (:conc-name :slot-prefix)
  (:constructor [make ext (ord-\*)])
  (:copier :copy-structure)
  (:include struct slot)
  (:type vector
    (slot int)
    type)
  (:print-object [o-printer]
    (predicate [print-function f-printer]
      :print-function f-printer arg)
    :type slot-type :read-only)
  :slot
  :doc)

- Define structure foo together with functions MAKE-foo, COPY-foo and foo\-P, and settable accessors foo\-slot. Instan-
cies are of class foo or, if defstruct option type is
given, of the specified type. They can be created by
(MAKE-foo [:slot value]) or, if ord-\* (see page 17) is given,
by (\make arg* [:key value]). In the latter case, arg* and
\keys correspond to the positional and keyword para-

ments in ord-\* whose vars in turn correspond to
slots. \printobject/print-function generate a \printobject
method for an instance bar of foo calling (o-printer bar
stream) or (f-printer bar stream print-level), respectively.
If :type without named is given, no foo\-P is created.

copy-structure structure
- Return copy of structure with shared slot values.

9 Control Structure

9.1 Predicates

(eq foo bar)  \(\) if foo and bar are identical.

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

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

(equalp foo bar)  \(\) if foo and bar are equivalent; or are the same character
ignoring case; or are numbers of the same value ignoring
type; or are equivalent pathnames; or are conses or arrays
of the same shape with equalp elements; or are structures
of the same type with equalp elements; or are hash-
tables of the same size with the same :test function, the same keys
in terms of :test function, and equalp elements.

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

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

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

(functionp foo)  \(\) if foo is of type function.
(fboundp (setf foo))
  ↷ T if foo is a global function or macro.

9.2 Variables

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

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

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

(setq symbol form)
  ↷ Set symbol's value cell to form. Deprecated.

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

(multiple-value-bind var values-form decl body-form)
  ↷ Evaluate body-forms with var s lexically bound to the return values of values-form. Return values of body-forms.

(progv symbols values form)
  ↷ Evaluate form s with locally established dynamic bindings of symbols to values or NIL. Return values of form.

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

(prog symbol values form)
  ↷ Evaluate forms with property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.

(let symbol key)
  ↷ First entry key from property list stored in symbol/in place, respectively, or default if there is no key. setfable.

(get symbol key [default])
  ↷ Return key and value of first entry from property-list matching a key from keys, and tail of property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.

(getf place key [default])
  ↷ Store value of foo in rightmost place shifting values of places left, returning first place.

(shiftf place+ foo)
  ↷ Rotate values of places left, old first becoming new last place's value. Return NIL.

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

(let [name value] ] (declare decl*) form)
  ↷ Evaluate forms with names lexically bound (in parallel or sequentially, respectively) to values. Return values of forms.
9.3 Functions

Below, ordinary lambda list \((\text{ord-}\lambda^*)\) has the form
\[
\texttt{[\text{var}^+ [\text{optional} \{ \texttt{[var \ (\text{init\text{-}\text{supplied-p}})]} \}] \ [\text{&rest} \ \text{var}]}
\]
\[
\text{\&key \ {var \ (t:key \ \text{var}) \ [\text{init\text{-}\text{supplied-p}]}}]
\]
\[
\text{\&allow\text{-}other\text{-}keys]] \ [\text{&aux} \ {var \ (\text{init\text{-}\text{supplied-p})}]}
\]
\]
\]
supplied-p is \(T\) if there is a corresponding argument. \text{init} forms can refer to any \text{init} and \text{supplied-p} to their left.

\[
\texttt{\{\text{defun} \ \text{foo} \ (\text{ord-}\lambda^*) \ (\text{setf} \ \text{foo}) \ (\text{new-value ord-}\lambda^*) \ (\text{declare} \ \text{decl}^*) \ [\text{doc}]}
\]
\[
\texttt{\{\text{lambda} \ (\text{ord-}\lambda^*) \ \text{form}\^*}
\]
\]
\]
\[
\text{\&declare \ (\text{ord-}\lambda^*) \ (\text{local-decl}^*) \ (\text{doc} \ \text{local-form}^*) \ (\text{declare} \ \text{decl}^*) \ \text{form}\^*}
\]
\]
\]
\[
\texttt{\{\text{let \ \{\text{labels}]] \ \{\text{foo} \ (\text{ord-}\lambda^*) \ (\text{setf} \ \text{foo}) \ (\text{new-value ord-}\lambda^*) \ (\text{declare} \ \text{local-decl}^*) \ (\text{doc} \ \text{local-form}^*) \ (\text{declare} \ \text{decl}^*) \ \text{form}\^*}
\]
\]
\]
\[
\texttt{\{\text{function} \ \text{foo} \ \{\text{lambda form}\^*}
\]
\]
\[
\texttt{\{\text{apply} \ \{\text{function} \ (\text{setf function}) \ \text{arg}^* \ \text{args} \}
\]
\]
\[
\texttt{\{\text{funcall function arg}^* \}
\]
\]
\[
\texttt{\{\text{multiple-value-call function form}\^* \}
\]
\]
\[
\texttt{\{\text{values-list list} \}
\]
\]
\[
\texttt{\{\text{values foo}^* \}
\]
\]
\[
\texttt{\{\text{multiple-value-list form} \}
\]
\]
\[
\texttt{\{\text{nth-value n form} \}
\]
\]
\[
\texttt{\{\text{complement function} \}
\]
\]
\[
\texttt{\{\text{constantly foo} \}
\]
\]
\[
\texttt{\{\text{identity foo} \}
\]
\[\text{\textbf{9.4 Macros}}\]

Below, macro lambda list (macro-\(\lambda\)) has the form of either
\[
\begin{align*}
\text{\&whole } \text{var} & | \text{E} | (\text{macro-}\lambda^*) | (\text{E}) \\
\text{\&optional} & \{ \text{var} | (\text{macro-}\lambda^*) \} \text{[init]} [\text{supplied-p}] \} | (\text{E}) \\
\text{\&rest} & (\text{rest-var}) | (\text{macro-}\lambda^*) | (\text{E}) \\
\text{\&key} & \{ \text{var} | (\text{macro-}\lambda^*) \} \text{[init]} [\text{supplied-p}] \} | (\text{E}) \\
\text{\&allow-other-keys} & \{ \text{var} | (\text{macro-}\lambda^*) \} \text{[init]} [\text{supplied-p}] \} | (\text{E}) \\
\text{\&aux} & (\text{var}) \text{[init]} [\text{supplied-p}] \} | (\text{E}) \\
\end{align*}
\]

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

\text{\(\text{\&defmacro}\)}
\text{\(\text{\&define-compiler-macro}\)}
\text{\(\text{\&define-symbol-macro}\)}
\text{\(\text{\&defsetf}\)}

Common Lisp Quick Reference

18
9.5 Control Flow

(if test then (else 𝑠))

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

(mcond (test then) (else 𝑠))

⇒ Return the values of the first then whose test returns T; return NIL if all tests return NIL.

(when (unless 𝑠) 𝑠)

⇒ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.
(\texttt{case \texttt{test}} \{ (\texttt{\textbackslash{key}}) \texttt{foo} \} (\texttt{\textbackslash{key}}) \texttt{bar} \})
\triangleright \text{Return the values of the first \texttt{foo} one of whose keys is \texttt{eq} \texttt{test}. Return values of \texttt{bar} if there is no matching key.}

(\texttt{\textsf{mcase} \texttt{test}} \{ (\texttt{\textbackslash{key}}) \texttt{foo} \} (\texttt{\textbackslash{key}}) \texttt{bar} \})
\triangleright \text{Return the values of the first \texttt{foo} one of whose keys is \texttt{eq} \texttt{test}. Signal non-correctable/correctable \texttt{type-error} if there is no matching key.}

(\texttt{\textsf{and \& form}})
\triangleright \text{Evaluate forms from left to right. Immediately return \texttt{NIL} if one form's value is \texttt{NIL}. Return values of last form otherwise.}

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

(\texttt{\textsf{progn form}})
\triangleright \text{Evaluate forms sequentially. Return values of last form.}

(\texttt{\textsf{multiple-value-prog1 form-r form}})
(\texttt{\textsf{prog1 form-r form}})
\triangleright \text{Evaluate forms in order. Return values/primary value, respectively, of form-r.}

(\texttt{\textsf{\textbackslash{tagbody} \{ \texttt{\textbackslash{tag}} \texttt{form} \}}})
\triangleright \text{Evaluate forms in a lexical environment. \texttt{tags} (symbols or integers) have lexical scope and dynamic extent, and are targets for \texttt{\textbackslash{go}}. Return \texttt{NIL}.}

(\texttt{\textsf{\textbackslash{go} \texttt{\textbackslash{tag}}}})
\triangleright \text{Within the innermost possible enclosing \texttt{\textbackslash{tagbody}}, jump to a tag, \texttt{eq} tag.}

(\texttt{\textsf{\textbackslash{catch} tag form}})
\triangleright \text{Evaluate forms and return their values unless interrupted by \texttt{\textbackslash{throw}}.}

(\texttt{\textsf{\textbackslash{throw} tag form}})
\triangleright \text{Have the nearest dynamically enclosing \texttt{\textbackslash{catch}} with a tag, \texttt{eq} tag return with the values of form.}

(\texttt{\texttt{\textbackslash{sleep} n}})
\triangleright \text{Wait n seconds; return \texttt{NIL}.}
9.6 Iteration

\[
\begin{align*}
&\{\textbf{do} \mid \textbf{do}\} \quad \{\textbf{var} \mid \textbf{start} \} \quad \{\textbf{step} \mid \textbf{form}\} \\
&\quad \{\textbf{tag} \mid \textbf{form}\} \\
&\quad \{\textbf{tagbody} \mid \textbf{body}\}
\end{align*}
\]

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

\[
\begin{align*}
&\{\textbf{dotimes} \mid \textbf{var}\} \quad \{\textbf{result} \mid \textbf{NIL}\} \\
&\quad \{\textbf{tag} \mid \textbf{form}\} \\
&\quad \{\textbf{tagbody} \mid \textbf{body}\}
\end{align*}
\]

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

\[
\begin{align*}
&\{\textbf{dolist} \mid \textbf{var}\} \quad \{\textbf{result} \mid \textbf{NIL}\} \\
&\quad \{\textbf{tag} \mid \textbf{form}\} \\
&\quad \{\textbf{tagbody} \mid \textbf{body}\}
\end{align*}
\]

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

9.7 Loop Facility

\[
\begin{align*}
&\{\textbf{loop} \mid \textbf{form}\} \\
&\quad \{\textbf{tag} \mid \textbf{form}\} \\
&\quad \{\textbf{tagbody} \mid \textbf{body}\}
\end{align*}
\]

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

\[
\begin{align*}
&\{\textbf{loop} \mid \textbf{clause}\} \\
&\quad \{\textbf{tag} \mid \textbf{form}\} \\
&\quad \{\textbf{tagbody} \mid \textbf{body}\}
\end{align*}
\]

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

d
amed \text{\texttt{name}}

\[
\begin{align*}
&\{\textbf{with} \mid \textbf{with}\} \quad \{\textbf{var-s} \mid \textbf{var-s}\} \\
&\quad \{\textbf{var-p} \mid \textbf{var-p}\} \\
&\quad \{\textbf{d-type} \mid \textbf{d-type}\} \\
&\quad \{\textbf{result} \mid \textbf{result}\}
\end{align*}
\]

\(\triangleright\) Give \textbf{loop}'s implicit \textbf{block} a name.

\[
\begin{align*}
&\{\textbf{for} \mid \textbf{as}\} \quad \{\textbf{var-s} \mid \textbf{var-s}\} \\
&\quad \{\textbf{var-p} \mid \textbf{var-p}\} \\
&\quad \{\textbf{d-type} \mid \textbf{d-type}\} \\
&\quad \{\textbf{result} \mid \textbf{result}\}
\end{align*}
\]

\(\triangleright\) Begin of iteration control clauses. Initialize and step (possibly trees of) local variables \textbf{var-s} sequentially and \textbf{var-p} in parallel.

\[
\begin{align*}
&\{\textbf{hash-key} \mid \textbf{hash-keys}\} \\
&\quad \{\textbf{hash-table} \mid \textbf{hash-table}\}
\end{align*}
\]

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

\[
\begin{align*}
&\{\textbf{hash-value} \mid \textbf{value}\}
\end{align*}
\]

\(\triangleright\) Bind \textbf{var} successively to the keys of \textbf{hash-table}; bind \textbf{value} to corresponding values.
Figure 1: Loop Facility, Overview.
(doform+)
  ➤ Evaluate forms in every iteration.

(if test i-clause and j-clause end)
  ➤ If test returns T, T, or NIL, respectively, evaluate i-clause and j-clauses; otherwise, evaluate k-clause and l-clauses.

it
  ➤ Inside i-clause or k-clause: value of test.

return formit
  ➤ Return immediately, skipping any finally parts, with values of form or it.

(collectformat)formit 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.

(appendappendinconc)formit 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.

(countcountingformit into n type)
  ➤ Count the number of times the value of form or it is T. If no n is given, count into an anonymous variable which is returned after termination.

(sumsummingformit into sum type)
  ➤ Calculate the sum of the primary values of form or it. If no sum is given, sum into an anonymous variable which is returned after termination.

(maximizemaximizingminimizeminimizingformit into max-min type)
  ➤ Determine the maximum or minimum, respectively, of the primary values of form or it. If no max-min is given, use an anonymous variable which is returned after termination.

(initiallyfinally)form+
  ➤ Evaluate forms before begin, or after end, respectively, of iterations.

repeat num
  ➤ Terminate aloop after num iterations; num is evaluated once.

(whileuntil)test
  ➤ Continue iteration until test returns NIL or T, respectively.

(alwaysnever)test
  ➤ Terminate aloop returning NIL and skipping any finally parts as soon as test is NIL or T, respectively. Otherwise continue aloop with its default return value set to T.

(thereis)test
  ➤ Terminate aloop when test is T and return value of test, skipping any finally parts. Otherwise continue aloop with its default return value set to NIL.

(aloopfinish)
  ➤ Terminate aloop 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.

(defclass foo (superclass*standard-object)

(slot (reader reader*)
(writer writer*)
(accessor accessor*)

(:allocation (instance :class instance)

(:initarg :initarg-name)
(:initform initarg)

(:documentation slot-doc)
)

(:default-initargs (name value*)
)

(:documentation class-doc)

(:metaclass name)

(slot-boundp f)

(slot-exists-p f)

(slot-makunbound f)

(make-instance class named foo)

(make-instance class named foo)

(find-class symbol (environment))

(reinitialize-instance instance (initarg value*)

(reinitialize-instance instance (other-keyarg)*)

(shared-initialize)

(slot-value foo slot) ⇒ Return value of slot in foo, set-fail.

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

(with-slots (var slot*)
(with-accessors (var accessor*)

(make-instances-obsolete)

(make-instances-obsolete)

(update-instance-for-different-class previous current)

(update-instance-for-redefined-class)

(initialize-instance instance)

(setf (class-name) new-name)

(setf (class-name) new-name)

(GET/set name of class)

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

(change-class instance new-class (other-keyarg*)

(change-class instance new-class (other-keyarg*)

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

(update-instance-class of change-class by means of shared-initialize.

24
10.2 Generic Functions

(defmethod-args (c-type c-arg) {ascalib-method-args})

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

(ensure-generic-function foo (setf foo) {standard-generic-function})

⇒ 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.

(mdefmethod foo (setf foo) {standard-method})

⇒ Before ;after :around qualifier method-class method-class method-combination c-type c-arg* lambda-list lambda-list environment environment

⇒ 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.

(next-method-p)

⇒ T if enclosing method has a next method.

(mdefgeneric foobar (setf foobar) {standard-method})

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

10.2 Generic Functions

(update-instance-for-redefined-class new-instance added-slots discarded-slots discarded-slots-property-list {standard-method})

⇒ On behalf of make-instancesobsolete 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.
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 forms. forms are enclosed in an implicit block foo. Applicable qualifiers depend on the method-combination type; see section 10.3.

{add-method, remove-method} generic-function-method
> Add (if necessary) or remove (if any) method to/from generic-function.

{find-method generic-function qualifiers specializers [error T]} generic-function-method
> Return suitable method, or signal error.

{compute-applicable-methods generic-function args} generic-function-method
> List of methods suitable for args, most specific first.

{call-next-method arg * current args *} generic-function-method
> From within a method, call next method with args; return its values.

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

{invalid-method-error method control arg *} generic-function-method
> Signal error on applicable method with invalid qualifiers, or on method combination. For control and args see format, page 36.

{no-next-method generic-function method arg *} generic-function-method
> Called on invocation of call-next-method when there is no next method. Default method signals error. Not to be called by user.

{function-keywords method} generic-function-method
> Return list of keyword parameters of method and T if other keys are allowed.

{method-qualifiers method} generic-function-method
> List of qualifiers of method.

10.3 Method Combination Types

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

and/or/append/list/nconc/progn/max/min/+ Simple built-in method-combination types; have the same usage as the c-types defined by the short form of define-method-combination.

{define-method-combination c-type
  :documentation string
  :operator c-type}
> Create new simple method-combination of the given c-type; the function method is called with as many methods as specified by the method combination. This function is used to define new method combinations that cannot be directly represented by the short form define-method-combination.
\( \text{Short Form. Define new method-combination c-type.} \)

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

\((\text{mdefine-method-combination c-type (ord-\lambda*) (group})
\begin{align*}
&\{\text{qualifier} \ast \ast\} \\
&\{\text{predicate}\} \\
&\{\text{description control}\} \\
&\{\text{order} \{\text{most-specific-first} \text{most-specific-last}\}\} \\
&\{\text{required bool}\} \\
&\{\text{:arguments method-combination-\lambda}\} \\
&\{\text{:generic-function symbol}\} \\
&\{\text{declare decl*\ast}\} \\
&\{\text{body\ast}\}\}
\end{align*}
\)

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

\((\text{mcall-method})
\begin{align*}
&\{\text{method}\} \\
&\{\text{(mmake-method form)}\} \\
&\{\text{next-method (mmake-method form)}\}
\end{align*}
\)

\( \text{From within an effective method form, call method with the arguments of the generic function and with information about its next-methods; return its values.} \)

\( \text{11 Conditions and Errors} \)

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

\((\text{mdefine-condition foo (parent-type}\text{condition-doc})
\begin{align*}
&\{\text{slot}\} \\
&\{\text{[:reader reader\ast]}\} \\
&\{\text{[:writer writer\ast]}\} \\
&\{\text{[:accessor accessor\ast]}\} \\
&\{\text{[:allocation instance class]}\} \\
&\{\text{[:initarg :initarg-name]}\} \\
&\{\text{:form type documentation slot-doc}\}
\end{align*}
\)

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

  (\text{signal} \{condition-condition-type \{\text{instance-name-value}\}\}
  \{\text{control} args\})
  \rightarrow Unless handled, signal as condition, warning or error, respectively, condition or a new instance of condition-type or, with \text{format} control and args (see page 36), simple-condition, simple-warning, or simple-error, respectively. From \text{signal} and \text{warn}, return \text{NIL}.

  (error \text{continue-control}
  \begin{align*}
  \text{condition continue-arg} & \text{condition-condition-type } \{\text{instance-name-value}\} \{\text{control} args\} \\
  \text{continue-control} & \text{condition-condition-type } \{\text{instance-name-value}\} \{\text{control} args\}
  \end{align*}
  \rightarrow Unless handled, signal as correctable error condition or a new instance of condition-type or, with \text{format} control and args (see page 36), \text{simple-error}. In the debugger, use \text{format} arguments \text{continue-control} and \text{continue-args} to tag the continue option. Return \text{NIL}.

  (\text{ignore-errors form})
  \rightarrow Return values of forms or, in case of errors, \text{NIL} and the condition.

  (\text{invoke-debugger} condition)
  \rightarrow Invoke debugger with condition.

  (\text{handler-case} foo
  \begin{align*}
  \text{type } \{\text{var}\} & \text{(declare decl\text{\text{-}ast}) condition-condition-type } \{\text{instance-name-value}\} \\
  \text{with-no-error} & \text{(\text{ord-}λ) (\text{declare decl\text{\text{-}ast}) form})
  \end{align*}
  \rightarrow If, on evaluation of \text{foo}, a condition of type is signalled, evaluate matching condition-forms with var bound to the condition, and return their values. Without a condition, bind ord-λ to values of foo and return values of forms or, without a \text{no-error} clause, return values of foo. See page 17 for (\text{ord-}λ).

  (\text{handler-bind} \{\text{condition-condition-type handler-function}\} form)
  \rightarrow Return values of forms after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.

  (\text{with-simple-restart} \{\text{restart \text{N I L}}\} control-condition-type form)
  \rightarrow Return values of forms unless restart is called during their evaluation. In this case, describe restart using \text{format} control and args (see page 36) and return \text{NIL} and \text{T}.

  (\text{restart-case form} \{\text{restart (ord-λ)}\}
  \begin{align*}
  \text{interactive arg-function} & \text{report-function}\{\text{string}}\{\text{variable}\} \\
  \text{test function}\{\text{string}\}
  \end{align*}
  \rightarrow Return values of form or, if during evaluation of form one of the dynamically established restarts is called, the values of its restart-forms. A restart is visible under condition if \text{funcall} \#\text{-test-function-condition} (condition) returns \text{T}. If presented in the debugger, restarts are described by \text{string} or by \#\text{-report-function} (of a stream). A restart can be called by (\text{invoke-restart restart args}), where args match ord-λ, or by (\text{invoke-restart-interactively restart}) where a list of the respective args is supplied by \#\text{-arg-function}. See page 17 for ord-λ.
(restart-bind ((restart restart-function
   \{interactive-function arg-function
   \{report-function report-function
   \{test-function test-function

   \})) form))

\>

Return values of forms evaluated with dynamically established restarts whose restart-functions should perform a non-local transfer of control. A restart is visible under condition if (test-function condition) returns T. If presented in the debugger, restarts are described by restart-function (of a stream). A restart can be called by (invoke-restart restart arg*), where args must be suitable for the corresponding restart-function, or by (invoke-restart-interactively restart) where a list of the respective args is supplied by arg-function.

(invoke-restart restart arg*)

(invoke-restart-interactively restart)

\>

Call function associated with restart with arguments given or prompted for, respectively. If restart function returns, return its values.

(find-restart
   \{compute-restarts name \[condition]\})

\>

Return innermost restart name, or a list of all restarts, respectively, out of those either associated with condition or un-associated at all; or, without condition, out of all restarts. Return NIL if search is unsuccessful.

(restart-name restart)

\>

Name of restart.

(with-condition-restarts condition restarts form)

\>

Evaluate forms with restarts dynamically associated with condition. Return values of forms.

(arithmetic-error-operation condition)

\>

List of function or of its operands respectively, used in the operation which caused condition.

(cell-error-name condition)

\>

Name of cell which caused condition.

(unbound-slot-instance condition)

\>

Instance with unbound slot which caused condition.

(print-not-readable-object condition)

\>

The object not readably printable under condition.

(package-error-package condition)

\>

Package, path, or stream, respectively, which caused the condition of indicated type.

(type-error-datum condition)

\>

Object which caused condition of type type-error, or its expected type, respectively.

(simple-condition-format-control condition)

\>

Return format control or list of format arguments, respectively, of condition.

*break-on-signals NIL

\>

Condition type debugger is to be invoked on.
12 Types and Classes

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

(typep foo type [environment]) ▷ T if foo is of type.

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

(the ident form) ▷ Declare values of form to be of type.

(coerce object type) ▷ Coerce object into type.

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

(if mtypecase mctypecase) foo [ident form] * ▷ Return values of the first form* whose type is foo of. Signal non-correctable/correctable type-error if no type matches.

(type-of foo) ▷ Type of foo.

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

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

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

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

(deftype foo (macro-λ) (declare doc form*) [environment])
▷ Define type foo which when referenced as (foo arg*) (or as foo if macro-λ doesn’t contain any required parameters) applies expanded forms to args returning the new type. For (macro-λ) see page 18 but with default value of * instead of NIL. forms are enclosed in an implicit block named foo.

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

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

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

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

(not type) ▷ Complement of type.

(and type* [environment]) ▷ Type specifier for intersection of types.

(or type* [environment]) ▷ Type specifier for union of types.

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

* ▷ As a type argument (cf. Figure 2): no restriction.
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 of indicated type.
- `(f input-stream-p stream)` ▷ Return T if stream is for input.
- `(f output-stream-p stream)` ▷ Return T if stream is for output.
- `(f interactive-stream-p stream)` ▷ Return T if stream is interactive.
- `(f open-stream-p stream)` ▷ Return 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.

13.2 Reader

- `(f y-or-n-p)` ▷ Ask user a question and return T or NIL depending on their answer.
- `(f yes-or-no-p)` ▷ Ask user a question and return T or NIL depending on their answer.
- `(f with-standard-io-syntax form)` ▷ Evaluate form s with standard behaviour of reader and printer. Return values of form s.
- `(f read)` ▷ Read printed representation of object.
- `(f read-from-string string)` ▷ Return object read from string and zero-indexed position of next character.
- `(f read-delimited-list char)` ▷ Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.
- `(f read-char)` ▷ Return next character from stream.
- `(f read-char-no-hang)` ▷ Next character from stream or NIL if none is available.
- `(f peek-char)` ▷ Next, or if mode is T, next non-whitespace character, or if mode is a character, next instance of it, from stream without removing it there.
- `(f unread-char character)` ▷ Put last read character back into stream; return NIL.
- `(f read-byte stream)` ▷ Read next byte from binary stream.
- `(f read-line stream)` ▷ 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 readable) ▷ Case sensitivity attribute (one of :upcase, :downcase, :preserve, :invert) of readable. setf.

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

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

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

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

13.3 Character Syntax

# multi-line-comment∗ | #
; one-line-comment∗
▷ Comments. There are stylistic conventions:
;;;; title ▷ Short title for a block of code.
;;; intro ▷ Description before a block of code.
;;; state ▷ State of program or of following code.
;;; explanation ▷ Regarding line on which it appears.
;;; continuation ▷

(foo* . bar) ▷ List of foos with the terminating cdr bar.

" ▷ Begin and end of a string.

'foo ▷ (quote foo); foo unevaluated.

'(foo) ▷ 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 ▷ (c character *c*), the character c.

#B: #O: n.; #:X: #:Rn ▷ Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.
Common Lisp Quick Reference

\[ \{\text{ratio } n \div d \} \text{ as short-float, single-float, double-float, long-float, or the type from } \text{read-default-float-format}. \]

\#C(a b) \text{ (} \text{complex a b}, \text{ the complex number } a + bi. \)

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

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

\#n\{\} \text{ n-dimensional array.} \]

\#S(\{type \{slot value\}\} \text{ Structure of type.} \]

\#P \text{ A pathname.} \]

\#:foo \text{ Uninterned symbol foo.} \]

\#:form \text{ Read-time value of form.} \]

\#\{read-eval\} \text{ If NIL, a reader-error is signalled at #,.} \]

\#integer= foo \text{ Give foo the label integer.} \]

\#integer# \text{ Object labelled integer.} \]

\#< \text{ Have the reader signal reader-error.} \]

\+#feature when-feature \text{ Means when-feature if feature is T; means unless-feature if feature is NIL. feature is a symbol from } \#\{features\}, \text{ or } \#\{and or \{feature \} \}. \]

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

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

13.4 Printer

\( \text{\{prin1 prin print \}} \text{ Print foo to stream, \text{readably, \text{readably} between a newline and a space, \text{readably} after a newline, or \text{human-readably} without any extra characters, respectively. prin1, prin and } \text{print return } \text{foo.} \]

\( \text{\{print-to-string \}} \text{ Print foo to } \text{string, \text{readably} or human-readably, respectively.} \]

\( \text{\{print-object \}} \text{ Print object to stream. Called by the Lisp printer.} \]

\( \text{\{print-unreadable-object \}} \text{ Enclosed in } \#< \text{ and } >, \text{ print } \text{foo by means of forms to stream. Return NIL.} \]

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

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

34
(write-char char stream)  
  Output char to stream.

(write-string string stream)  
  Write string to stream without/with a trailing newline.

(write-byte byte stream)  
  Write byte to binary stream.

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

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

(print-fill stream foo)  
  Eval parameter forms, which should print list, with stream locally bound to a pretty printing stream which outputs to the original stream. If list is in fact not a list, it is printed by write. Return NIL. Usable with /format directive ~f.

(print-tabular stream foo)  
  With stream bound to a pretty printing stream, print foo. If foo is a list, print as many elements per line as possible; do the same in a table with a column width of n ems; or print either all elements on one line or each on its own line, respectively. Return NIL.

(print-dispatch stream foo)  
  Eval parameter forms, which should print list, with stream locally bound to a pretty printing stream which outputs to the original stream. If list is in fact not a list, it is printed by write. Return NIL.

(print-circle stream foo)  
  Indicate list, which should print list, as the last element of a pretty printing block. Returns NIL.

(print-logical-block stream list)  
  Eval parameter forms, which should print list, with stream locally bound to a pretty printing stream which outputs to the original stream. If list is in fact not a list, it is printed by write. Return NIL.

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

(print-tab stream c)  
  Move cursor forward to column number c + ki, k ≥ 0 being as small as possible.

(print-indent block)  
  Specify indentation for innermost logical block relative to leftmost position/to current position. Return NIL.

(print-exit-if-list-exhausted)  
  If list is empty, terminate logical block. Return NIL otherwise.
(\pprint-newline  (linear \fill \miser \mandatory) \stream  \standard-output)
  \> Print a conditional newline if \stream \> a pretty printing stream. Return \NIL.

\*\pprint-array*  \> If T, print arrays \readably.

\*\pprint-base*  \(\V\)  \> Radix for printing rationals, from 2 to 36.

\*\pprint-case*  \(\V\)  \> Prefix symbol names all uppercase (\upcase), all lowercase (\downcase), capitalized (\capitalize).

\*\pprint-circle*  \(\V\)  \> If T, avoid indefinite recursion while printing circular structure.

\*\pprint-escape*  \(\V\)  \> If \NIL, do not print escape characters and package prefixes.

\*\pprint-gensym*  \(\V\)  \> If T, print \#: before uninterned symbols.

\*\pprint-length*  \(\V\)  \> If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

\*\pprint-miser-width*  \(\V\)  \> If integer and greater than the width available for printing a substructure, switch to the more compact miser style.

\*\pprint-pretty*  \(\V\)  \> If T, print prettily.

\*\pprint-radix*  \(\V\)  \> If T, print rationals with a radix indicator.

\*\pprint-readably*  \(\V\)  \> If T, print \readably or signal error \print-not-readable.

\*\pprint-right-margin*  \(\V\)  \> Right margin width in ems while pretty-printing.

(\set-\pprint-dispatch  type  function  [\priority]  \table \*\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 \*\pprint-dispatch*)
  \> Return highest priority \function associated with type of \foo and \IF there was a matching \type specifier in \table. If \function is \NIL, return \NIL.

(\copy-\pprint-dispatch  \table \*\pprint-dispatch*)
  \> Return copy of \table or, if \table is \NIL, initial value of \*\pprint-dispatch.

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

13.5 Format

(\formatter  control)
  \> Return function of \stream and \string applying \format to \stream, \control, and \string returning \NIL or any excess \args.

(\format  \(*\NIL\)  \out-string  \out-stream  \control  \string*)
  \> 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 \string. 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.
Justification. Justify text produced by `texts` in a field of at least `min-col` columns. With `;`, right justify; with `@`, left justify. If this would leave less than `space` characters on the current line, output `nl-text` first.

|- `[-prefix < prefix-line-prefix -suffix]` |- `body - ;; suffix ; -prefix -suffix` |

Logical Block. Act like `print-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 `print-pop`. With `;`, prefix and suffix default to `(and)` and `(and)`. When closed by `~@:`, spaces in `body` are replaced with conditional newlines.

Indent. Set indentation to `n` relative to leftmost/to current position.

Tabulate. Move cursor forward to column number `c+ki`, `k ≥ 0` being as small as possible. With `;`, calculate column numbers relative to the immediately enclosing section. With `@`, move to column number `c0 + c+ki` 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.

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.

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.

Call Function. Call all-uppercase `package`:`function` with the arguments stream, `format-argument`, `colon-p`, and `prefix`es for printing `format-argument`.

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) ➤ Open file-stream to path.

(\make-concatenated-stream input-stream*) ➤ Return stream of indicated type.

(\make-two-way-stream input-stream-part output-stream-part) ➤ Return a string-stream supplying the characters from string.

(\make-synonym-stream variable-bound-to-stream) ➤ Return a string-stream accepting characters (available via \get-output-stream-string).

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

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

(\synonym-stream-symbol synonym-stream) ➤ Return symbol of synonym-stream.

(\get-output-stream-string string start end) ➤ Clear and return as a string characters on string-stream.

(\file-position stream start end position) ➤ Return position within stream, or set it to position and return T on success.

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

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

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

(\clear-output) ➤ End output to stream and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.
(close stream [aborts booms])
➤ Close stream. Return T if stream had been open. If abort
is T, delete associated file.

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

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

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

(with-output-to-string (foo string) [element-type] [up: upspec]) [declare decl]* form)
➤ Evaluate forms with foo locally bound to an output string-stream. Append output to string and return values of forms if string is given. Return string containing output otherwise.

(stream-external-format stream)
➤ External file format designer.

*:terminal-io  ➤ Bidirectional stream to user terminal.
*:standard-input  ➤ Standard input stream.
*:standard-output  ➤ Standard output stream.
*:error-output  ➤ Standard error output stream, respectively.
*:debug-io*  ➤ Bidirectional streams for debugging and user interaction.
*:query-io*  ➤

13.7 Pathnames and Files

(makepathname

{host [host nil unspecif] device [device nil unspecif]}
directory [directory nil unspecif] wild-inferiors)

{name [file-name wild nil unspecif] type [file-type wild nil unspecif] version [version wild nil unspecif] defaults path from [remote pathname-string]}

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

{pathname-host
pathname-device
pathname-directory
pathname-name
pathname-type
pathname-version path-or-stream)

➤ Return pathname component.
(parse-namestring foo [host]
[default-pathname *default-pathname-defaults*]

[(start string (end nil) [junk-allowed booแนวทาง]])]
> Return pathname converted from string, pathname, or
stream foo; and position where parsing stopped.

(merge-pathnames path-or-stream
[default-path-or-stream *default-pathname-defaults*]
[default-version :newest])
> Return pathname made by filling in components missing
in path-or-stream from default-path-or-stream.

*default-pathname-defaults*
> Pathname to use if one is needed and none supplied.

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

(enough-namestring path-or-stream
[root-path *default-pathname-defaults*]
> Return minimal path string that sufficiently describes the
path of path-or-stream relative to root-path.

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

(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.

(pathname path-or-stream)
> Pathname of path-or-stream.

(logical-pathname logical-path-or-stream)
> Logical pathname of logical-path-or-stream. Logi-
cal pathnames are represented as all-uppercase
"[host:]{{dir}*{name}*{type}*}{LISP}.{version}*{NEWEST}".

(logical-pathname-translations logical-host)
> List of (from-wildcard to-wildcard) translations for
logical-host. setfable.

(load-logical-pathname-translations logical-host)
> Load logical-host’s translations. Return NIL if already
loaded; return T if successful.

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

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

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

(file-author file)
> Return name of file owner.

(file-length stream)
> Return length of stream.

(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.

(delete-file file)
> Delete file. Return T.

(directory path)
> List of pathnames matching path.
14 Packages and Symbols

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

14.1 Predicates

\( \text{symbolp} \ foo \) \( \implies \) T if foo is of indicated type.

\( \text{packagep} \ foo \) \( \implies \) Keyword, evaluates to package.

\( \text{keywordp} \ foo \) \( \implies \) Possibly unexported symbol of package.

\( \text{defpackage} \ foo \) \( \implies \) Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and shd-symbols. Add shd-symbols to foo’s shadowing list.

\( \text{make-package} \ foo \) \( \implies \) Create package foo.

\( \text{rename-package} \ foo \) \( \implies \) Rename package. Return renamed package.

\( \text{in-package} \foo \) \( \implies \) Make package foo current.

\( \text{use-package} \) other-packages \( \implies \) Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.

\( \text{package-use-list} \) other-packages \( \implies \) List of other packages used by/using package.

\( \text{delete-package} \) \( \implies \) Delete package. Return T if successful.

\( \text{list-all-packages} \) \( \implies \) The current package.

\( \text{package-name} \) \( \implies \) List of registered packages.

\( \text{package-name} \) \( \implies \) Name of package.

\( \text{package-nicknames} \) \( \implies \) Nicknames of package.

\( \text{find-package} \ name \) \( \implies \) Package with name (case-sensitive).

\( \text{find-all-symbols} \ foo \) \( \implies \) List of symbols foo from all registered packages.
Common Lisp Quick Reference

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
  ▷ Return fresh, uninterned symbol #<n> with n from *gensym-counters*. Increment *gensym-counters*.

gentemp
  ▷ Intern fresh symbol in package. Deprecated.

copy-symbol
  ▷ Return uninterned copy of symbol. If props is T, give copy the same value, function and property list.
Common Lisp Quick Reference

(symbol-name symbol)  (symbol-package symbol)  (symbol-plist symbol)  (symbol-value symbol)  (symbol-function symbol)

▶ Name, package, property list, value, or function, respectively, of symbol. setfable.

(documentation (setf documentation) new-doc foo)

▶ Get/set documentation string of foo of given type.

t

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

nil

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

14.4 Standard Packages

common-lisp cl

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

common-lisp-user cl-user

▶ Current package after startup; uses package common-lisp.

keyword

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

15 Compiler

15.1 Predicates

(special-operator-p foo)  (compiled-function-p foo)

▶ T if foo is a special operator.

▶ T if foo is of type compiled-function.

15.2 Compilation

(compile NIL definition name (setf name) [definition])

▶ Return compiled function or replace name’s function definition with the compiled function. Return T in case of warnings or errors, and T in case of warnings or errors excluding style-warnings.

(compile-file path [verbose bool] [print bool] [external-format file-format] [if-does-not-exist bool] [output-file path])

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

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

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

(load path [verbose bool] [print bool] [load-print] [if-does-not-exist bool] [external-format file-format] [if-does-not-exist bool])

▶ Load source file or compiled file into Lisp environment. Return T if successful.
Common Lisp Quick Reference

- `*compile-file* pathname true-name` → Input file used by `*compile-file*`/`load`.
- `*compile* (print* `verbose*` → Defaults used by `*compile-file*`/`load`.
- `*load*` →
  - `(:compile-toplevel `compile`)` → 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.)
  - `(:load-toplevel `load`)` →
  - `(:execute `eval`)` → Return values of forms if `s` `eval-when` is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere, respectively. Return `NIL` if forms are not evaluated. (`compile`, `load` and `eval` deprecated.)
  - `:(write t `eval-when`)` →
- `globally decl* form*` → Evaluate `form*`s in a lexical environment with declarations `decl*` in effect. Return values of `form*`s.
- `with-compilation-unit (`override bool `NIL`) form*` → Return values of `form*`s. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of `form*`s.
- `load-time-value `quote` `read-only `NIL` form` → Evaluate `form` at compile time and treat its value as literal at run time.
- `quote foo` → Return unevaluated `foo`.
- `make-load-form foo `environment` `NIL` form` → 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` → Return a creation form and an initialization form which on evaluation construct an object equivalent to `foo` with `slots` initialized with the corresponding values from `foo`.
- `macro-function symbol `environment` `NIL` name form` → Return specified macro function, or compiler macro function, respectively, if any. Return `NIL` otherwise. `setf`able.
- `eval` → Return values of value of `arg` evaluated in global environment.

15.3 REPL and Debugging

++, ++, +++ → Form currently being evaluated by the REPL.
- `apropos string `package` `NIL` form` → Print interned symbols containing `string`.
- `apropos-list string` `package` `NIL` form → List of interned symbols containing `string`.
- `dribble `path` form` → Save a record of interactive session to file at `path`. Without `path`, close that file.
- `ed` `file-or-function` `NIL` → Invoke editor if possible.
\[\text{macroexpand-1 form environment} \]

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

\[\text{macroexpand-hook} \]

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

\[\text{trace function } \]

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

\[\text{untrace function } \]

- Stop functions, or each currently traced function, from being traced.

\[\text{trace-output } \]

- Output stream \text{trace} and \text{time} send their output to.

\[\text{step form } \]

- Step through evaluation of form. Return values of form.

\[\text{break control arg} \]

- Jump directly into debugger; return NIL. See page 36, \text{format}, for control and args.

\[\text{time form } \]

- Evaluate forms and print timing information to \text{trace-output}. Return values of form.

\[\text{inspect foo} \]

- Interactively give information about foo.

\[\text{describe foo } \]

- Send information about foo to stream. Called by \text{describe}.

\[\text{describe-object foo } \]

- Send information about foo to stream. Called by \text{describe}.

\[\text{disassemble function} \]

- Send disassembled representation of function to \text{standard-output}. Return NIL.

\[\text{room NIL default} \]

- Print information about internal storage management to \text{standard-output}.

### 15.4 Declarations

\[\text{proclaim decl} \]

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

\[\text{declare decl} \]

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

\[\text{declaration foo} \]

- Make foos names of declarations.

\[\text{dynamic-extent variable* (function function)*} \]

- Declare lifetime of variables and/or functions to end when control leaves enclosing block.

\[\text{ftype type variable*} \]

- Declare variables or functions to be of type.

\[\text{ignore} \]

- Suppress warnings about used/unused bindings.
Common Lisp Quick Reference

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

(optimize
\[
\begin{array}{c}
\text{compilation-speed} \\
\text{debug} \\
\text{safety} \\
\text{space} \\
\text{speed}
\end{array}
\]
)
▷ Tell compiler how to optimize. \( n = 0 \) means unimportant, \( n = 1 \) is neutral, \( n = 3 \) means important.

(special var)
▷ Declare vars to be dynamic.

16 External Environment

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

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

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

(decode-universal-time universal-time [time-zone curr])
(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)
(software)
(machine)
▷ Name or version of implementation, operating system, or hardware, respectively.

(machine-instance)
▷ Computer name.

47