(\text{\texttt{(\texttt{sinh}\ \texttt{a})}}) \quad \triangleright \quad \text{sinh}\ a, \ \text{cosh}\ a, \text{ or } \text{tanh}\ a, \text{ respectively.}

(\text{\texttt{(\texttt{cosh}\ \texttt{a})}})

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

(\text{\texttt{(\texttt{cis}\ \texttt{a})}}) \quad \triangleright \quad \text{Return } e^{i\alpha} = \cos a + i \sin a.

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

(\text{\texttt{(\texttt{max}\ \texttt{num}^+})}}) \quad \triangleright \quad \text{Greatest or least, respectively, of } \text{num}'s.

(\text{\texttt{(\texttt{round}\\ \texttt{fround})}}) \quad n \ [d]

\begin{align*}
\begin{cases}
\text{\texttt{(\texttt{round}\\ \texttt{fround})}} \\
\text{\texttt{(\texttt{floor}\\ \texttt{ffloor})}} \\
\text{\texttt{(\texttt{ceiling}\\ \texttt{fceiling})}} \\
\text{\texttt{(\texttt{truncate}\\ \texttt{ftruncate})}}
\end{cases}
\end{align*}

\begin{itemize}
\item \triangleright \text{Return as integer or float, respectively, } n/d \text{ rounded, or rounded towards } -\infty, +\infty, \text{ or } 0, \text{ respectively; and remainder.}
\end{itemize}

(\text{\texttt{(\texttt{mod}\\ \texttt{rem})}}) \quad n \ d

\begin{itemize}
\item \triangleright \text{Same as } \text{floor} \text{ or } \text{truncate}, \text{ respectively, but return remainder only.}
\end{itemize}

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

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

(\#\texttt{*random-state*}) \quad \triangleright \text{Current random state.}

(\text{\texttt{(\texttt{float-sign}\ \texttt{num-a}\ [\texttt{num-b}]})}} \quad \triangleright \text{num-b with num-a's sign.}

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

(\text{\texttt{(\texttt{numerator}\ \texttt{rational})}})

(\text{\texttt{(\texttt{denominator}\ \texttt{rational})}})

\begin{itemize}
\item \triangleright \text{Numerator or denominator, respectively, of } rational's canonical form.\end{itemize}

(\text{\texttt{(\texttt{realpart}\ \texttt{number})}})

(\text{\texttt{(\texttt{imagpart}\ \texttt{number})}})

\begin{itemize}
\item \triangleright \text{Real part or imaginary part, respectively, of } number.\end{itemize}

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

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

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

(\text{\texttt{(\texttt{rational}\ \texttt{real})}})

(\text{\texttt{(\texttt{rationalize}\ \texttt{real})}})

\begin{itemize}
\item \triangleright \text{Convert real to rational. Assume complete/limited accuracy for real.}\end{itemize}

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

1 Numbers 3 9.5 Control Flow . . . 21
1.1 Predicates . . . . 3 9.6 Iteration . . . . 22
1.2 Numeric Functions . 3 9.7 Loop Facility . . . . 22
1.3 Logic Functions . 5 10 CLOS 25
1.4 Integer Functions . 6 10.1 Classes . . . . . 25
1.5 Implementation- 10.2 Generic Functions . 26
Dependent . . . . . 10.3 Method Combi-
nation Types . . . . 28
2 Characters 7
3 Strings 8 11 Conditions and Errors 28
4 Conses 8 12 Types and Classes 31
4.1 Predicates . . . . 8
4.2 Lists . . . . 9
4.3 Association Lists 10
4.4 Trees . . . . . 10
4.5 Sets . . . . . 11
5 Arrays 11
5.1 Predicates . . . . 11
5.2 Array Functions . . . . . 11
5.3 Vector Functions . . . . . 12
6 Sequences 12
6.1 Seq. Predicates . . . . . 12
6.2 Seq. Functions . . . . . 13
7 Hash Tables 15
8 Structures 16
8.1 Predicates . . . . . 16
8.2 Variables . . . . . 17
8.3 Functions . . . . . 18
8.4 Macros . . . . . 19
16 External Environment 49

1 Numbers

1.1 Predicates

(i = number+)  ▷ \text{T} if all numbers, or none, respectively, are equal in value.
(i > number+)  ▷ \text{T} if numbers are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.
(i >= number+)  ▷ \text{T} if numbers are monotonically decreasing or monotonically non-increasing.
(i < number+)  ▷ \text{T} if numbers are monotonically increasing, or monotonically non-decreasing.
(i <= number+)  ▷ \text{T} if numbers are monotonically non-decreasing.

1.2 Numeric Functions

(i+ a b)  ▷ \text{T} if a < 0, a = 0, or a > 0, respectively.
(i* a b)  ▷ \text{T} if \text{int} is even odd, respectively.
(i+ Int)  ▷ \text{T} if \text{int} is of indicated type.
(i+ Complex)  ▷ \text{T} if \text{foo} is of indicated type.
(i+ Random-state-p)  ▷ \text{T} if foo is of indicated type.

Typographic Conventions

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

them  ▷ Placeholder for actual code.
me  ▷ Literal text.
[foo]  ▷ Either one foo or nothing; defaults to bar.
foo*; {foo}  ▷ Zero or more foos.
foo+; {foo}  ▷ One or more foos.
foos  ▷ English plural denotes a list argument.
{foo|bar|baz}; {foo|bar|baz}  ▷ Either foo, or bar, or baz.
{foo|bar|baz}  ▷ Anything from none to each of foo, bar, and baz.

foo  ▷ Argument foo is not evaluated.
bar  ▷ Argument bar is possibly modified.
foo^b  ▷ foo* is evaluated as in \text{progn}; see page 21.
foo; bar; baz  ▷ Primary, secondary, and nth return value.
\text{T}; \text{NIL}  ▷ t, or truth in general; and \text{nil} or ().
3 Strings

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

\[(\text{string} \quad \text{foo})\] \[\Rightarrow \text{T} \text{ if foo is of indicated type.}\]

\[(\text{simple-string-p} \quad \text{foo})\]

\[\{\text{string=} \quad \text{foo bar}\]

\[\text{string= }\]

\[\{\text{start1 start-foo }\]

\[\text{start2 start-bar }\]

\[\text{end1 end-foo }\]

\[\text{end2 end-bar }\]

\[\Rightarrow \text{Return T if subsequences of foo and bar are equal. Obey/ignore, respectively, case.}\]

\[\text{string} /= \]

\[\text{string} > \]

\[\text{string} >= \]

\[\text{string} < \]

\[\text{string} <= \]

\[\Rightarrow \text{If foo is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in foo. Otherwise return NIL. Obey/ignore, respectively, case.}\]

\[(\text{make-string} \quad \text{size}\]

\[\{\text{initial-element char}\]

\[\text{element-type type}\]

\[\text{character}\}

\[\Rightarrow \text{Return string of length size.}\]

\[(\text{string} \quad x)\]

\[\text{string-capitalize}\]

\[\text{string-upcase}\]

\[\text{string-downcase}\]

\[\Rightarrow \text{Convert } x \text{ (symbol, string, or character) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.}\]

\[\{\text{nstring-capitalize}\]

\[\text{nstring-upcase}\]

\[\text{nstring-downcase}\]

\[\Rightarrow \text{Convert string into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.}\]

\[\text{string-trim}\]

\[\text{string-left-trim}\]

\[\text{string-right-trim}\]

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

\[\text{char string i}\]

\[\text{schar string i}\]

\[\Rightarrow \text{Return zero-indexed i-th character of string ignoring/obeying, respectively, fill pointer. setfable.}\]

\[\text{parse-integer string}\]

\[\{\text{start start-i}\]

\[\text{end end-i}\]

\[\text{radix int}\]

\[\text{junk-allowed bool}\]

\[\Rightarrow \text{Return integer parsed from string and index of parse end.}\]

4 Conses

4.1 Predicates

\[(\text{consp} \quad \text{foo})\]

\[\Rightarrow \text{Return T if foo is of indicated type.}\]

\[(\text{listp} \quad \text{foo})\]

\[\Rightarrow \text{Return T if } \text{foo} \text{ is NIL.}\]

\[(\text{endp} \quad \text{list})\]

\[\Rightarrow \text{Return T if } \text{list}/\text{foo} \text{ is NIL.}\]

1.3 Logic Functions

Negative integers are used in two’s complement representation.

\[(\text{boole operation int-a int-b})\]

\[\Rightarrow \text{Return value of bitwise logical operation. operations are}\]

\[\text{boole-1} \]

\[\Rightarrow \text{int-a.}\]

\[\text{boole-2} \]

\[\Rightarrow \text{int-b.}\]

\[\text{boole-cl} \]

\[\Rightarrow \text{~int-a.}\]

\[\text{boole-c2} \]

\[\Rightarrow \text{~int-b.}\]

\[\text{boole-set} \]

\[\Rightarrow \text{All bits set.}\]

\[\text{boole-clr} \]

\[\Rightarrow \text{All bits zero.}\]

\[\text{boole-eqv} \]

\[\Rightarrow \text{int-a } \equiv \text{ int-b.}\]

\[\text{boole-and} \]

\[\Rightarrow \text{int-a } \land \text{ int-b.}\]

\[\text{boole-andc1} \]

\[\Rightarrow \text{~int-a } \land \text{ int-b.}\]

\[\text{boole-andc2} \]

\[\Rightarrow \text{int-a } \land \text{ ~int-b.}\]

\[\text{boole-nand} \]

\[\Rightarrow \text{~(int-a } \land \text{ int-b).}\]

\[\text{boole-ior} \]

\[\Rightarrow \text{int-a } \lor \text{ int-b.}\]

\[\text{boole-orc1} \]

\[\Rightarrow \text{~int-a } \land \text{ int-b.}\]

\[\text{boole-orc2} \]

\[\Rightarrow \text{int-a } \land \text{ ~int-b.}\]

\[\text{boole-xor} \]

\[\Rightarrow \text{~int-a } \lor \text{ int-b.}\]

\[\text{boole-nor} \]

\[\Rightarrow \text{~int-a } \lor \text{ int-b.}\]

\[(\text{lognot} \quad \text{integer})\]

\[\Rightarrow \text{~integer.}\]

\[(\text{logeqv} \quad \text{integer})\]

\[\Rightarrow \text{~integer.}\]

\[(\text{logand} \quad \text{integer})\]

\[\Rightarrow \text{~integer.}\]

\[(\text{logandc1} \quad \text{int-a int-b})\]

\[\Rightarrow \text{~int-a } \land \text{ int-b.}\]

\[(\text{logandc2} \quad \text{int-a int-b})\]

\[\Rightarrow \text{int-a } \land \text{ ~int-b.}\]

\[(\text{lognand} \quad \text{int-a int-b})\]

\[\Rightarrow \text{~(int-a } \land \text{ int-b).}\]

\[(\text{logxor} \quad \text{integer})\]

\[\Rightarrow \text{~integer.}\]

\[(\text{logior} \quad \text{integer})\]

\[\Rightarrow \text{~integer.}\]

\[(\text{parse-integer} \quad \text{string}\]

\[\{\text{start start-i}\]

\[\text{end end-i}\]

\[\text{radix int}\]

\[\text{junk-allowed bool}\]

\[\Rightarrow \text{Return integer parsed from string and index of parse end.}\]

\[(\text{logbitp} \quad \text{i int})\]

\[\Rightarrow \text{T if zero-indexed i-th bit of int is set.}\]

\[(\text{logtest} \quad \text{int-a int-b})\]

\[\Rightarrow \text{Return T if there is any bit set in int-a which is set in int-b as well.}\]

\[(\text{logcount} \quad \text{int})\]

\[\Rightarrow \text{Number of 1 bits in int } \geq 0, \text{ number of 0 bits in int } < 0.\]
1.4 Integer Functions

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

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

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

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

(ldb byte-spec integer)  
▷ Extract byte denoted by byte-spec from integer. setable.

(deposit-field)  
▷ Extract byte denoted by byte-spec from integer. setable.

(dbp int-a byte-spec int-b)  
▷ Return int-b with bits denoted by byte-spec replaced by corresponding bits of int-a, or by the low (byte-size byte-spec) bits of int-a, respectively.

(mask-field byte-spec integer)  
▷ Return copy of integer with all bits unset but those denoted by byte-spec. setable.

(byte size position)  
▷ Byte specifier for a byte of size bits starting at a weight of

(byte-size byte-spec)  
▷ Size or position, respectively, of byte-spec.

(value-byte-position)  
▷ Value of byte position.

1.5 Implementation-Dependent

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

(single-float)  
▷ Available numbers closest to −0 or +0, respectively.

(double-float)  
▷ Available numbers closest to −∞ or +∞, respectively.

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

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

2 Characters

The standard-char type comprises a-z, k-Z, 0-9, Newline, Space, and !?$*.,;+*/-\~<=>#%\@[]().

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

(standard-char-p char)  
▷ T if argument is of indicated type.

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

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

(lower-case-p character)  
▷ T if all characters, or none, respectively, are equal.

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

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

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

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

(char-not-equal character+)  
▷ T if all characters, or none, respectively, are equal.

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

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

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

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

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

(char-name char)  
▷ char’s name if any, or NIL.

(name-char foo)  
▷ Character named foo if any, or NIL.

(char-int character)  
▷ Code of character.

(char-code character)  
▷ Character with code.

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

(character c)  
▷ Return #\c.
6 Sequences

6.1 Sequence Predicates

\((\textit{every})\) [\textit{test sequence}] \\
\((\textit{not every})\) \\
\(\xrightarrow{\text{Return} \ \textit{NIL} \ \text{or} \ \textit{T}, \ \text{respectively, as soon as} \ \textit{test} \ \text{on any set of corresponding elements of} \ \textit{sequences} \ \text{returns} \ \textit{NIL}.}\)

\((\textit{some})\) [\textit{test sequence}] \\
\((\textit{not any})\) \\
\(\xrightarrow{\text{Return} \ \textit{NIL} \ \text{or} \ \textit{T}, \ \text{respectively, as soon as} \ \textit{test} \ \text{on any set of corresponding elements of} \ \textit{sequences} \ \text{returns} \ \textit{T}.}\)

4.2 Lists

\((\textit{cons})\) [\textit{foo} \ \textit{bar}] \\
\((\textit{null})\) \\
\((\textit{list})\) [\textit{foo}+] \\
\((\textit{list*})\) [\textit{foo}] \\
\((\textit{nil})\) \\
\(\xrightarrow{\text{Return list of} \ \textit{foo} \ \text{becoming} \ \text{cdr} \ \text{of} \ \text{last} \ \text{foo}.}\) \\
\(\xrightarrow{\text{Return} \ \text{if only one} \ \text{foo} \ \text{given}.}\)

\((\textit{make-list})\) [\textit{num} \ \textit{[initial-element} \ \textit{foo}]) \\
\xrightarrow{\text{New list with} \ \textit{num} \ \text{elements set to} \ \textit{foo}.}\)

\((\textit{list-length})\) [\textit{list}] \\
\xrightarrow{\text{Length of} \ \textit{list}; \ \textit{NIL} \ \text{for} \ \textit{circular} \ \textit{list}.}\)

\((\textit{car})\) [\textit{list}] \\
\xrightarrow{\text{Car of} \ \textit{list} \ \text{or} \ \textit{NIL} \ \text{if} \ \textit{list} \ \text{is} \ \textit{NIL}. \ \textit{setf} \ \text{able}.}\)

\((\textit{cdr})\) [\textit{list}] \\
\xrightarrow{\text{Cdr of} \ \textit{list} \ \text{or} \ \textit{NIL} \ \text{if} \ \textit{list} \ \text{is} \ \textit{NIL}. \ \textit{setf} \ \text{able}.}\)

\((\textit{nthcdr})\) [\textit{n} \ \textit{list}] \\
\xrightarrow{\text{Return} \ \text{tail of} \ \textit{list} \ \text{after} \ \text{calling} \ \textit{cdr} \ \text{n} \ \text{times}.}\)

\((\textit{first})\) \ldots \((\textit{nth})\) \textit{list} \\
\xrightarrow{\text{Return} \ \textit{nth} \ \text{element of} \ \textit{list} \ \text{if} \ \text{any}, \ \text{or} \ \textit{NIL} \ \text{otherwise}. \ \textit{setf} \ \text{able}.}\)

\((\textit{first})\) \textit{list} [\textit{num}] \\
\xrightarrow{\text{Return} \ \text{list of} \ \text{last} \ \text{num} \ \text{conses} \ \text{of} \ \textit{list}.}\)

\((\textit{butlast})\) [\textit{list}] \\
\xrightarrow{\text{Return} \ \text{list} \ \text{excluding} \ \text{last} \ \text{num} \ \text{conses}.}\)

\((\textit{rplaca})\) [\textit{cons} \ \textit{object}] \\
\xrightarrow{\text{Replace} \ \text{car}, \ \text{cdr} \ \text{respectively, of} \ \text{cons} \ \text{with} \ \text{object}.}\)

\((\textit{adiff})\) [\textit{list}] \\
\xrightarrow{\text{If} \ \textit{foo} \ \text{is} \ \text{tail} \ \text{of} \ \text{list}, \ \text{return} \ \text{preceding} \ \text{part} \ \text{of} \ \text{list}. \ \text{Otherwise} \ \text{return} \ \text{list}.}\)

\((\textit{adjoin})\) [\textit{foo} \ \textit{list}] \\
\xrightarrow{\text{Return} \ \text{list} \ \text{if} \ \text{foo} \ \text{is} \ \text{already} \ \text{member} \ \text{of} \ \textit{list}. \ \text{If} \ \text{not}, \ \text{return} \ \text{foo} \ \text{list}.}\)

\((\textit{adiff})\) [\textit{list} \ \textit{foo}] \\
\xrightarrow{\text{Return} \ \text{list} \ \text{if} \ \text{foo} \ \text{is} \ \text{already} \ \text{member} \ \text{of} \ \textit{list}. \ \text{If} \ \text{not}, \ \text{return} \ \text{foo} \ \text{list}.}\)

\((\textit{adiff})\) [\textit{list} \ \textit{foo}] \\
\xrightarrow{\text{Return} \ \text{list} \ \text{if} \ \text{foo} \ \text{is} \ \text{already} \ \text{member} \ \text{of} \ \textit{list}. \ \text{If} \ \text{not}, \ \text{return} \ \text{foo} \ \text{list}.}\)
4.3 Association Lists

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

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

(assoc foo alist)  ▷ Return alist if foo exists in alist.

(nassoc foo alist)  ▷ Return alist if foo does not exist in alist.

(assoc-if-not test alist [key function])  ▷ First cons whose car, or cdr, respectively, satisfies test.

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

4.4 Trees

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

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

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

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

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

5.2 Array Functions

(make-array dimension-sizes [adjustable bool])  ▷ Return fresh, or readjust, respectively, of lists.

(array-has-fill-pointer-p array)  ▷ Return T if array is adjustable/has a fill pointer, respectively.

(array-in-bounds-p array [subscripts])  ▷ Return T if subscripts are in array’s bounds.

5.1 Predicates

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

(vectorp foo)  ▷ T if foo is a vector.

(simple-vector-p foo)  ▷ T if foo is a simple vector.

(bit-vector-p foo)  ▷ T if foo is a bit vector.

(adjustable-array-p array)  ▷ T if array is adjustable/has a fill pointer, respectively.

(array-has-fill-pointer-p array)  ▷ Return T if array is adjustable/has a fill pointer, respectively.

(array-in-bounds-p array [subscripts])  ▷ Return T if subscripts are in array’s bounds.

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

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

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

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

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

(defun foo)
  (defstruct foo
    (:conc-name :functor-foo)
    (:constructors '(foo-ctor)
                  (copy-ctor copy-ctor)
                  (copier copier)
    )
    (include struct
      (slot :init :type :read-only b)
    )
    (funcall slot)
  )

(defun copy-structure structure)
  (copy structure)

(defun make-structure structure-type size [initial-element foo])
  (make-structure structure-type size)

(defun copy-structure structure)
  (copy structure)

9 Control Structure

9.1 Predicates

(defun eq foo bar)
  (eq foo bar)

(defun eql foo bar)
  (eql foo bar)

(defun equal foo bar)
  (equal foo bar)

(defun eqal foo bar)
  (eqal foo bar)

(defun not foo)
  (not foo)

(defun boundp symbol)
  (boundp symbol)

6.2 Sequence Functions

(defun make-sequence sequence-type size [initial-element foo])
  (make-sequence sequence-type size)

(defun concatenate sequence-a sequence-b)
  (concatenate sequence-a sequence-b)

(defun merge sequence-a sequence-b test [key function])
  (merge sequence-a sequence-b test)

(defun fill sequence foo)
  (fill sequence foo)

(defun count sequence)
  (count sequence)

(defun count-if sequence)
  (count-if sequence)

(defun count-if-not sequence)
  (count-if-not sequence)

(defun reverse sequence)
  (reverse sequence)

(defun stable-sort sequence test [key function])
  (stable-sort sequence test)

(defun reverse sequence)
  (reverse sequence)

(defun find position sequence)
  (find position sequence)

(defun test function)
  (test function)

(defun test-not function)
  (test-not function)

(defun end function)
  (end function)

(defun start function)
  (start function)

(defun end function)
  (end function)

(defun start function)
  (start function)

(defun end function)
  (end function)

(defun start function)
  (start function)

(defun end function)
  (end function)

(defun start function)
  (start function)

(defun end function)
  (end function)

(defun start function)
  (start function)

(defun end function)
  (end function)
\begin{itemize}
\item \textbf{find-if \textit{test sequence}}
\item \textbf{find-if-not \textit{test sequence}}
\item \textbf{position-if \textit{test sequence}}
\item \textbf{position-if-not \textit{test sequence}}
\end{itemize}

\begin{itemize}
\item Return first element in \textit{sequence} which satisfies \textit{test}, or its position relative to the begin of \textit{sequence}, respectively.
\end{itemize}

\begin{itemize}
\item \textbf{search \textit{sequence-a sequence-b}}
\end{itemize}

\begin{itemize}
\item Search \textit{sequence-b} for a subsequence matching \textit{sequence-a}. Return position in \textit{sequence-b}, or \text{NIL}.
\end{itemize}

\begin{itemize}
\item \textbf{remove \textit{foo sequence}}
\item \textbf{delete \textit{foo sequence}}
\end{itemize}

\begin{itemize}
\item Make copy of \textit{sequence} without elements matching \textit{foo}.
\end{itemize}

\begin{itemize}
\item \textbf{remove-if \textit{test sequence}}
\item \textbf{delete-if \textit{test sequence}}
\item \textbf{remove-if-not \textit{test sequence}}
\item \textbf{delete-if-not \textit{test sequence}}
\end{itemize}

\begin{itemize}
\item Make copy of \textit{sequence} with all (or \textit{count}) elements satisfying \textit{test} removed.
\end{itemize}

\begin{itemize}
\item \textbf{remove-duplicates \textit{sequence}}
\item \textbf{delete-duplicates \textit{sequence}}
\end{itemize}

\begin{itemize}
\item Make copy of \textit{sequence} without duplicates.
\end{itemize}

\begin{itemize}
\item \textbf{substitute \textit{new old sequence}}
\item \textbf{nsubstitute \textit{new old sequence}}
\item \textbf{substitute-if \textit{new test sequence}}
\item \textbf{nsubstitute-if \textit{new test sequence}}
\end{itemize}

\begin{itemize}
\item Make copy of \textit{sequence} with all (or \textit{count}) olds replaced by \textit{new}.
\end{itemize}

\begin{itemize}
\item \textbf{replace \textit{sequence-a sequence-b}}
\end{itemize}

\begin{itemize}
\item Replace elements of \textit{sequence-a} with elements of \textit{sequence-b}.
\end{itemize}

\begin{itemize}
\item \textbf{map \textit{type function sequence}}
\end{itemize}

\begin{itemize}
\item Apply \textit{function} successively to corresponding elements of the \textit{sequences}. Return values as a \textit{sequence} of \textit{type}. If \textit{type} is \text{NIL}, return \text{NIL}.
\end{itemize}

\section{7 Hash Tables}

The Loop Facility provides additional hash table-related functionality; see \texttt{loop}, page 22. Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 10 and 17.

\begin{itemize}
\item \textbf{hash-table-p \textit{foo}}
\item \textbf{make-hash-table \textit{size int-rehash-size num-rehash-threshold num}}
\item \textbf{gethash \textit{key hash-table} \texttt{[default \text{NIL}]} \textit{key} \text{NIL}}
\item \textbf{hash-table-count \textit{hash-table}}
\item \textbf{remhash \textit{key hash-table}}
\item \textbf{maphash \textit{function hash-table}}
\item \textbf{with-hash-table-iterator \textit{foo hash-table} \texttt{[declare \textit{decl}]} \textit{form}}
\item \textbf{hash-table-test \textit{hash-table}}
\item \textbf{hash-table-size \textit{hash-table}}
\item \textbf{hash-table-rehash-size \textit{hash-table}}
\item \textbf{hash-table-rehash-threshold \textit{hash-table}}
\item \textbf{sexhash \textit{foo}}
\end{itemize}
(symbol-macrolet ((foo expansion-form*) (declare decl*) form*)
  \(\triangleright\) Evaluate forms with locally defined symbol macros foo.

(defsetf function updater [doc]
  ((self-\(\lambda\)) (s-var*)
   (declare decl*) [doc] form*)
where defsetf lambda list (self-\(\lambda\)) has the form
\(\{\text{var} \text{[optional]} \text{var} \text{[rest var]} \text{key} \text{[allow-other-keys]} \text{environment var}\}\)
\(\triangleright\) Specify how to setf a place accessed by function.
Short form: (setf (function arg*) value-form) is replaced by
\((\text{updater arg* value-form});\) the latter must return value-form.
Long form: on invocation of (setf (function arg*) value-form),
forms must expand into code that sets the place accessed
where self-\(\lambda\) and s-var* describe the arguments of function
and the value(s) to be stored, respectively; and that returns the value(s) of s-var*.
forms are enclosed in an implicit \(\text{block}\) named function.

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

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

(define-modify-macro foo ([optional]
  \([\text{var} \text{[initvar \(\text{[supplied-p]}\)}]} \text{[rest var]} \text{function [doc]}\])
\(\triangleright\) Define macro foo able to modify a place. On invocation of (foo place arg*),
the value of function applied to place and args will be stored into place and returned.

lambda-list-keywords
\(\triangleright\) List of macro lambda list keywords. These are at least:

&whole var \(\triangleright\) Bind var to the entire macro call form.

&optional var* \(\triangleright\) Bind vars to corresponding arguments if any.

&rest [\body\] var \(\triangleright\) Bind var to a list of remaining arguments.

&key var* \(\triangleright\) Bind vars to corresponding keyword arguments.

&allow-other-keys \(\triangleright\) Suppress keyword argument checking. Callers can do so using \\(\text{allow-other-keys} \text{T}\).

&environment var \(\triangleright\) Bind var to the lexical compilation environment.

&aux var* \(\triangleright\) Bind vars as in \(\text{let}\).

9.2 Variables

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

(defparameter function foo [form [setf foo]])
\(\triangleright\) Unless bound already, assign value of form to dynamic variable foo.

(setf function foo [form])
\(\triangleright\) Set places to primary values of forms. Return values of last form/NIL; work sequentially/in parallel, respectively.

(setq function [symbol form]*)
\(\triangleright\) Set symbols to primary values of forms. Return value of last form/NIL; work sequentially/in parallel, respectively.

(setf function symbol foo)
\(\triangleright\) Set symbol’s value cell to foo. Deprecated.

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

(mshift function place+ foo)
\(\triangleright\) Store value of foo in rightmost place shifting values of places left, returning first place.

(mrotatf function)
\(\triangleright\) Rotate values of places left, old first becoming new last place’s value. Return NIL.

(mmakunbound function)
\(\triangleright\) Delete special variable foo if any.

(gget symbol key [default nil])
\(\triangleright\) First entry key from property list stored in symbol/in place, respectively, or \text{default} if there is no key. settable.

(gget-properties function list keys)
\(\triangleright\) 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.

(gremprop symbol key)
\(\triangleright\) Remove first entry key from property list stored in symbol/in place, respectively. Return \text{T} if key was there, or NIL otherwise.

(gprogv function symbols values form*)
\(\triangleright\) Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.

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

Below, ordinary lambda list (ord-\(\lambda^*\)) has the form
\[
(\text{var}^* \text{[optional} \text{var} \text{[init]} \text{[supplied-p]}]) \text{[&key} \text{var}\text{]} \text{[&aux} \text{var} \text{[init]} \text{[supplied-p]}]) \text{[&allow-other-keys]}\]

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

\[
(\text{mdefun} \text{foo} \text{(ord-}\lambda^*) \text{[new-value ord-}\lambda^*) \text{[doc]} \text{[declare decl]}^*) \text{[&key} \text{var}\text{]} \text{[&aux} \text{var} \text{[init]} \text{[supplied-p]}]) \text{[&allow-other-keys]}\]

\[
(\text{mlambda} \text{ord-}\lambda^*) \text{form}^*\]

Define a function named foo or (setf foo), or an anonymous function, respectively, which apply forms to ord-\(\lambda\). For mdefun, forms are enclosed in an implicit block named foo.

\[
(\text{flet} \text{labels} \text{[&key} \text{var} \text{[init]} \text{[supplied-p]}]) \text{[&allow-other-keys]}\]

\[
(\text{funcall} \text{function} \text{arg}^*) \text{[&allow-other-keys]}\]

\[
(\text{multiple-value-call} \text{function} \text{form}^*)\]

\[
(\text{values-list} \text{list})\]

\[
(\text{values} \text{foo}^*)\]

\[
(\text{values} \text{foo}^*)\]

\[
(\text{multiple-value-list} \text{form})\]

\[
(\text{nth-value} \text{n} \text{form})\]

\[
(\text{complement} \text{function})\]

9.4 Macros

Below, macro lambda list (macro-\(\lambda^*\)) has the form of either
\[
([\text{&whole} \text{var} [\text{E} \text{var}] \text{[macro-}\lambda^*) \text{[E]}\]

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

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

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

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.

\[
(\text{define-macro} \text{foo} \text{[macro-}\lambda^*) \text{[declare decl]}^*) \text{[doc] form}^*\]

\[
(\text{define-compiler-macro} \text{foo} \text{[setf foo]} \text{[macro-}\lambda^*) \text{[declare decl]}^*) \text{[doc] form}^*\]

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

\[
(\text{macrolet} ([\text{foo} \text{macro-}\lambda^*) \text{[declare decl]}^*) \text{[doc] macro-form}^*) \text{[declare decl]}^*) \text{form}^*\]

Evaluate forms with locally defined mutually invisible macros foo which are enclosed in implicit blocks of the same name.
= foo [then bar]\[\]
  \> Bind var initially to foo and later to bar.

across vector
  \> Bind var to successive elements of vector.

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

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

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

{symbol|symbols|present-symbol|present-symbols| external-symbol|external-symbols} \{of\[\] in\} package
  \> Bind var successively to the accessible symbols, or the present symbols, or the external symbols respectively, of package.

{do|doing} form\+
  \> Evaluate forms in every iteration.

{if|when|unless} test \{i-clause \{and\ j-clause\}* \{else k-clause \{and\ l-clause\}*\}\ [end]
  \> If test returns T, T, or NIL, respectively, evaluate i-clause and j-clauses; otherwise, evaluate k-clause and l-clauses.

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

return {form|it}
  \> Return immediately, skipping any finally parts, with values of form or it.

{collect|collecting} \{form|it\} \{into\ list\}
  \> Collect values of form or it into list. If no list is given, collect into an anonymous list which is returned after termination.

{append|appending|nconc|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 \#loop after num iterations; num is evaluated once.

{while|until} test
  \> Continue iteration until test returns NIL or T, respectively.
(catch (tag form))
  ▷ Evaluate forms and return their values unless interrupted by ,throw.

(throw tag form)
  ▷ Have the nearest dynamically enclosing ,catch with a tag of eq tag return with the values of form.

(sleep n)
  ▷ Wait n seconds; return NIL.

9.6 Iteration

\[
\begin{align*}
&\{\text{do} \{\text{m} \text{do}\}\} (\text{var} \text{ [start [step]]}) (\text{stop result}) (\text{declare decl})^+ \\
&\text{tagbody} (\text{form})^+ \\
&\text{tagbody} (\text{form})^+
\end{align*}
\]

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

( dotimes (var i [result]) (declare decl) ^+ (tag form)^+ )
  ▷ Evaluate ,tagbody-like body with var successively bound to integers from 0 to i – 1. Upon evaluation of result, var is i. Implicitly, the whole form is a ,block named NIL.

(dolist (var list [result]) (declare decl) ^+ (tag form)^+ )
  ▷ Evaluate ,tagbody-like body with var successively bound to the elements of list. Upon evaluation of result, var is NIL. Implicitly, the whole form is a ,block named NIL.

9.7 Loop Facility

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

(,loop clause^+)
  ▷ Loop Facility. For Loop Facility keywords see below and Figure 1.

\begin{align*}
\text{named \{\text{loop}\} } &\text{ ▷ Give ,loop’s implicit ,block a name.} \\
&\{\text{with} \{\text{var-s} \text{ [var-s]}\} \{\text{d-type} \text{ [foo]}\}^+ \\
&\text{[and} \{\text{var-p} \text{ [var-p]}\} \{\text{d-type} \text{ [bar]}\}^+ \\
&\text{where destructuring type specifier d-type has the form} \\
&\{\text{fixnum float NIL of-type \{[type]\}^+}\} \\
&\text{▷ Initialize (possibly trees of) local variables var-s sequentially and var-p in parallel.} \\
&\{\text{for} \{\text{as} \{\text{var-s} \text{ [var-s]}\} \{\text{d-type} \text{ [foo]}\}^+ \\
&\text{[and} \{\text{var-p} \text{ [var-p]}\} \{\text{d-type} \text{ [bar]}\}^+ \\
&\text{▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables var-s sequentially and var-p in parallel. Destructuring type specifier d-type as with with.} \\
&\text{[from} \text{downfrom} \text{start} \text{to} \text{downto} \text{above} \text{down} \text{above} \text{form} \\
&\text{▷ Start stepping with start} \\
&\text{[upto downto} \text{below} \text{above} \text{form} \\
&\text{▷ Specify form as the end value for stepping.} \\
&\text{[in] \text{list} \\
&\text{▷ Bind var to successive elements/tails, respectively, of list.} \\
&\text{[step] \text{function} \{\text{car}\} \\
&\text{▷ Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.}
\end{align*}

Figure 1: Loop Facility, Overview.
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|append|list|conc|prog|max|min+
- Simple built-in method-combination types; have the same usage as the c-types defined by the short form of defmethod-combination.

(defun make-method-combination (c-type)
  (list :documentation string :identity-with-one-argument bool)
  > 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 function. From within this method, :call-next-method can call less specific :around methods if there are any. If not, or if there are no :around methods at all, return from the calling :call-next-method or from the generic function, respectively, the values of (primary-method gen-arg*)*, gen-arg* being the arguments of the generic function. The primary-methods are ordered (most-specific-first) (most-specific-last) (specified as c-arg in defgeneric). Using c-type as the qualifier in defmethod makes the method primary.

(defun make-method-combination (c-type (ord-λ*)) ((group
  (qualifier* [#]))
  (predicates)
  (description control)
  (order (most-specific-first) (most-specific-last))
  (required bool)
  (method-combination-λ*)
  (generic-function symbol)
  (declare decl*)
  (doc)
  > Long Form. Define new method-combination c-type. A call to a generic function using c-type will be equivalent to a call to the forms returned by body* with ord-λ* bound to c-arg* (cf. defgeneric), with symbol bound to the generic function, with method-combination-λ* bound to the arguments of the generic function, and with groups bound to lists of methods. An applicable method becomes a member of the leftmost group whose predicate or qualifiers match. Methods can be called via call-method. Lambda lists (ord-λ*) and (method-combination-λ*) according to ord-λ on page 18, the latter enhanced by an optional &whole argument.

(defun call-method
  (method (make-method-form))
  ((next-method (make-method-form))
  > 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.

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 32.
Common Lisp Quick Reference

(change-class instance new-class {initarg values} 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.

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

(initialize-instance instance )  
(update-instance-for-different-class previous current)  
{initarg values} other-keyarg*)  
▷ Set slots on behalf of make-instance of change-class by means of shared-initialize.

(update-instance-for-redefined-class new-instance added-slots  
discarded-slots discarded-slots-property-list {initarg values}  
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.

(allocate-instance class {initarg values} other-keyarg*)  
▷ Return uninitialized instance of class. Called by make-instance.

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

(slot-missing class instance slot )  
(slot-boundp slot-makunbound 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.

10.2 Generic Functions

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

(defgeneric foo (required-var* &optional var* &rest var) &key (key var) &allow-other-keys)  
▷ 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. deffmethod-args resemble those of defmethod. For c-type see section 10.3.

(ensure-generic-function foo (setf foo))  
▷ List of methods usable for method-definition.
Figure 2: Precedence Order of System Classes ( ), Classes ( ),
Types ( ), and Condition Types ( ).
Every type is also a supertype of NIL, the empty type.
(with-simple-restart (restart \{ control arg*\} form\#)  
  \(\triangleright\) Return values of \texttt{forms} unless \texttt{restart} is called during their evaluation. In this case, describe \texttt{restart} using \texttt{format control} and \texttt{args} (see page 38) and return \texttt{NIL} and \texttt{T}.

(with-restart-case form (restart (ord-\*\#)  
  \{interactive arg-function\} \{report \texttt{report-function} \{string \texttt{string-function}\} \{test \texttt{test-function}\})  
  \(\triangleright\) Return values of \texttt{form} or, if during evaluation of \texttt{form} one of the dynamically established \texttt{restarts} is called, the values of its \texttt{restart-forms}. A \texttt{restart} is visible under \texttt{condition} if \texttt{(funcall \#$\texttt{test-function-condition} \texttt{condition})} returns \texttt{T}. If presented in the debugger, \texttt{restarts} are described by \texttt{string} or by \texttt{\#$\texttt{report-function}} (of a stream). A \texttt{restart} can be called by \texttt{(invoke-restart \texttt{restart arg*})}, where \texttt{args} match \texttt{ord-\*\#}; or by \texttt{(invoke-restart-interactively \texttt{restart})} where a list of the respective \texttt{args} is supplied by \texttt{arg-function}. See page 18 for \texttt{ord-\*\#}.

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

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

(find-restart \{compute-restarts \texttt{name} \texttt{condition}\}  
  \(\triangleright\) Return innermost \texttt{restart name}, or a list of all \texttt{restarts}, respectively, out of those either associated with \texttt{condition} or un-associated at all; or, without \texttt{condition}, out of all \texttt{restarts}. Return \texttt{NIL} if search is unsuccessful.

(restart-name \texttt{restart})  
  \(\triangleright\) Name of \texttt{restart}.

\begin{itemize}
  \item \texttt{abort}
  \item \texttt{muffle-warning}
  \item \texttt{continue}
  \item \texttt{store-value \texttt{value}}
  \item \texttt{use-value \texttt{value}}
\end{itemize}

  \(\triangleright\) Transfer control to innermost applicable \texttt{restart} with same name (i.e. \texttt{abort}, \texttt{continue} . . .) out of those either associated with \texttt{condition} or un-associated at all; or, without \texttt{condition}, out of all \texttt{restarts}. If no \texttt{restart} is found, signal \texttt{control-error} for \texttt{abort} and \texttt{muffle-warning}, or return \texttt{NIL} for the rest.

(with-condition-restarts \texttt{condition} \texttt{restarts form\#})  
  \(\triangleright\) Evaluate \texttt{forms} with \texttt{restarts} dynamically associated with \texttt{condition}. Return values of \texttt{forms}.

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

\begin{itemize}
  \item \texttt{(cell-error-name \texttt{condition})}  
    \(\triangleright\) Name of cell which caused \texttt{condition}.
  \item \texttt{(unbound-slot-instance \texttt{condition})}  
    \(\triangleright\) Instance with unbound slot which caused \texttt{condition}.
  \item \texttt{(print-not-readable-object \texttt{condition})}  
    \(\triangleright\) The object not readable printable under \texttt{condition}.
  \item \texttt{(package-error-package \texttt{condition})}  
  \item \texttt{(file-error-pathname \texttt{condition})}  
  \item \texttt{(stream-error-stream \texttt{condition})}  
    \(\triangleright\) Package, path, or stream, respectively, which caused the \texttt{condition} of indicated type.
  \item \texttt{(type-error-datum \texttt{condition})}  
  \item \texttt{(type-error-expected-type \texttt{condition})}  
    \(\triangleright\) Object which caused condition of type \texttt{type-error}, or its expected type, respectively.
  \item \texttt{(simple-condition-format-control \texttt{condition})}  
  \item \texttt{(simple-condition-format-arguments \texttt{condition})}  
    \(\triangleright\) Return \texttt{format control} or list of \texttt{format arguments}, respectively, of \texttt{condition}.
\end{itemize}

*break-on-signals*\texttt{\#t}  
  \(\triangleright\) Condition type debugger is to be invoked on.

*debugger-hook*\texttt{\#t}  
  \(\triangleright\) Function of condition and function itself. Called before debugger.

\section{Types and Classes}

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

\begin{itemize}
  \item \texttt{(typep \texttt{foo} \texttt{type} \{environment\})}  
    \(\triangleright\) \texttt{T} if \texttt{foo} is of \texttt{type}.
  \item \texttt{(subtypep \texttt{type-a} \texttt{type-b} \{environment\})}  
    \(\triangleright\) Return \texttt{T} if \texttt{type-a} is a recognizable subtype of \texttt{type-b}, and \texttt{NIL} if the relationship could not be determined.
  \item \texttt{(the \texttt{type} \texttt{form})}  
    \(\triangleright\) Declare values of \texttt{form} to be of \texttt{type}.
  \item \texttt{(coerce \texttt{object} \texttt{type})}  
    \(\triangleright\) Coerce \texttt{object} into \texttt{type}.
  \item \texttt{(mtypecase \texttt{foo} (\texttt{type a-form\#}) \(\{\text{otherwise} \texttt{T} \texttt{b-form\#}\})\))  
    \(\triangleright\) Return values of the first \texttt{a-form\#}'s whose \texttt{type} is \texttt{foo}. Return values of \texttt{b-forms} if no \texttt{type} matches.
  \item \texttt{(etypecase} \texttt{foo} (\texttt{type form\#})\))  
    \(\triangleright\) Return values of the first \texttt{form\#}'s whose \texttt{type} is \texttt{foo}. Signal non-correctable/correctable \texttt{type-error} if no \texttt{type} matches.
  \item \texttt{(type-of \texttt{foo})}  
    \(\triangleright\) \texttt{Type} of \texttt{foo}.
  \item \texttt{(mcheck-type \texttt{place} \texttt{type} \{string \texttt{\#t} \texttt{type}\})\))  
    \(\triangleright\) Signal correctable \texttt{type-error} if \texttt{place} is not of \texttt{type}. Return \texttt{NIL}.
  \item \texttt{(stream-element-type \texttt{stream})}  
    \(\triangleright\) \texttt{Type} of \texttt{stream} objects.
  \item \texttt{(array-element-type \texttt{array})}  
    \(\triangleright\) Element type \texttt{array} can hold.
  \item \texttt{(upgraded-array-element-type \texttt{type} \{environment\})\))  
    \(\triangleright\) Element type of most specialized array capable of holding elements of \texttt{type}.
\end{itemize}
13.4 Printer

(print1 foo [stream \standard-output])
> Print foo to stream, readable, readable between a newline and a space, readable after a newline, or human-readable without any extra characters, respectively. print1, print and princ return foo.

(print-to-string foo)
> Specify a type comprising foo or foos.

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

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

(not type)
> Complement of type.

(and type* m)
> Type specifier for intersection of types.

(or type*" m)
> 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.

13 Input/Output

13.1 Predicates

(streamp foo)
> T if foo is of indicated type.

(pathnamep foo)
> T if foo is a symbol from or directory.

(readablep foo)
> T if foo reads with standard behaviour of reader and printer.

(input-stream-p stream)
> Read elements of stream.

(output-stream-p stream)
> Output a newline to stream. Return NIL.

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

(pathname-match-p path wildcard)
> T if path matches wildcard.

(wildpathname-p path [{:host | :device | :directory | :name | :type} :version NIL])
> Return T if indicated component in path is wildcard. (NIL indicates any component.)

13.2 Reader

(y-or-n-p)
> Ask user a question and return T or NIL depending on their answer. See page 38, format, for control and args.

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

(read form)
> Read printed representation of object.

(read-preserving-whitespace form)
> Read form preserving whitespace.

(read-from-string string [eof-error \ [eof-error]] [\preserve-whitespace \ [\recursive \]]])
> Return object read from string and zero-indexed position of next character.
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.

(foo* [ . bar]) ⊲ List of foos 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 "c"), the character c.

#\n: #O n; #o n; #r n ⊲ Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.

n/d ⊲ The ratio 1/n.

{[m].n|[S|F|D|O|L|E]x}[0-9]|{[m]+[n] |[S|F|D|O|L|E]x} ⊲ m.n·10^n as short-float, single-float, double-float, long-float, or the type from *read-default-float-format*.

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

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

#**sequence ⊲ n-dimensional array.

#(n)foo* ⊲ Vector of some (or n) foos filled with last foo if necessary.

#(n)~b* ⊲ Bit vector of some (or n) bs filled with last b if necessary.

#S(type {slot value}) ⊲ Structure of type.

#Pstring ⊲ A pathname.

#:foo ⊲ Uninterned symbol foo.

#:form ⊲ Read-time value of form.

.:*read-eval* ⊲ If nil, a reader-error is signalled at #:.

#integer = foo ⊲ Give foo the label integer.

#:integer ⊲ Object labelled integer.

#:<> ⊲ Have the reader signal reader-error.
\{- [\mathbb{N}] ; \rightarrow [\mathbb{N}] \}
\quad \triangleright \text{Indent. Set indentation to } n \text{ relative to leftmost/to current position.}
\hline
\{- [\mathbb{M}] \mid [\mathbb{M}] \rightarrow [\mathbb{M}] \}
\quad \triangleright \text{Tabulate. Move cursor forward to column number } c + ki, k \geq 0 \text{ being as small as possible. With } ;\text{ calculate column numbers relative to the immediately enclosing section. With } \emptyset \text{, move to column number } c_0 + c + ki \text{ where } c_0 \text{ is the current position.}

\{- [\mathbb{M}] \mid [\mathbb{M}] \rightarrow [\mathbb{M}] \emptyset \}
\quad \triangleright \text{Go-To. Jump } m \text{ arguments forward, or backward, or to argument } n.
\hline
\{- [\mathbb{M}] \mid [\mathbb{M}] \rightarrow [\mathbb{M}] \emptyset \}
\quad \triangleright \text{Iteration. Use } \text{text} \text{ repeatedly, up to } \text{limit}, \text{ as control string for the elements of the list argument or (with } \emptyset \text{) for the remaining arguments. With } ; \text{ or } \emptyset, \text{ list elements or remaining arguments should be lists of which a new one is used at each iteration step.}
\hline
\{- [\mathbb{E}] [\mathbb{E}] \}
\quad \triangleright \text{Escape Upward. Leave immediately } <\rightarrow, <\longleftarrow, <\rightarrow>, <\longleftarrow, -?, \text{ or the entire } \text{format} \text{ operation. With one to three prefixes, act only if } x = 0, x = y, \text{ or } x \leq y \leq z, \text{ respectively.}
\hline
\{- [\mathbb{M}] [\mathbb{M}] [\mathbb{M}] \}
\quad \triangleright \text{Conditional Expression. Use the zero-indexed argument (or } i\text{th if given) } \text{text} \text{ as a } \text{if} \text{ format control subclause. With } ; \text{ use the first text if the argument value is } \text{NIL}, \text{ or the second text if } \text{it is } T. \text{ With } \emptyset, \text{ do nothing for an argument value of } \text{NIL}. \text{ Use the only } \text{text} \text{ and leave the argument to be read again if it is } T.
\hline
\{- [\mathbb{M}] \emptyset \}
\quad \triangleright \text{Recursive Processing. Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original argument.}
\hline
\{- [\mathbb{M}] \ emptyset \}
\quad \triangleright \text{Call Function. Call all-} \text{upper} \text{-case } \text{package} \text{-function} \text{ with the arguments stream, format-argument, colon-p, at-sign-p and prefixes} \text{ for printing format-argument.}
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{Write. Print argument of any type obeying every printer control variable. With } ;\text{, pretty-print. With } \emptyset, \text{ print without limits on length or depth.}
\hline
\{- [\mathbb{M}] \emptyset \}
\quad \triangleright \text{In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.}

13.6 Streams
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{Open path to } \text{path}.
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{Close path to } \text{path}.
\hline
\{- [\mathbb{M}] [\mathbb{M}] \}
\quad \triangleright \text{Rename path to } \text{path}.
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{Remove path.}
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{Move cursor forward to column number } c + ki, k \geq 0 \text{ being as small as possible.}
\hline
\{- [\mathbb{M}] [\mathbb{M}] \}
\quad \triangleright \text{Specify indentation for innermost logical block relative to leftmost position/to current position. Return } \text{NIL} \text{ otherwise.}
\hline
\{- [\mathbb{M}] [\mathbb{M}] \}
\quad \triangleright \text{Move cursor forward to column number } c + ki, k \geq 0 \text{ being as small as possible.}
\hline
\{- [\mathbb{M}] [\mathbb{M}] \}
\quad \triangleright \text{Print a conditional newline if } \text{stream} \text{ is a pretty printing stream. Return } \text{NIL}
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{If } T, \text{ print arrays } \text{readably.}
\hline
\{- [\mathbb{M}] \}
\quad \triangleright \text{Radix for printing rationals, from 2 to 36.
print-case
- Print symbol names all uppercase (upcase), all lowercase (downcase), capitalized (capitalize).

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

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

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

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

print-miser-width
- If integer and greater than the width available for printing a substructure, switch to the more compact miser style.

print-pretty
- If T, print prettyly.

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

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

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

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

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

(copy-print-dispatch) [table]
- Return copy of table or, if table is NIL, initial value of *print-print-dispatch*.

(print-print-dispatch)
- Current pretty print dispatch table.

13.5 Format

(formatter)
- 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-stream and arg*. Output to out-string, out-stream or, if first argument is T, to *standard-output*. Return NIL. If first argument is NIL, return formatted output.

- [min-col (col-inc [min-pad [*pad-char*]])]
  - [i] [0] [S]
  - Aesthetic/Standard. Print argument of any type for consumption by humans/by the reader, respectively. With i, print NIL as () rather than NIL; with 0, add pad-chars on the left rather than the right.

- [radix [width [*pad-char*] [comma-char
  - [comma-interval]]] [i] [0] R]
  - Radix. (With one or more prefix arguments.) Print argument as number; with i, group digits comma-interval each; with 0, always prepend a sign.

{-R} [-R] [-OR] [-R]}
- 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.

- [width [*pad-char*] [comma-char
  - [comma-interval]]] [i] [0] [D][E][X]
- Decimal/Binary/Octal/Hexadecimal. Print integer argument as number. With i, group digits comma-interval each; with 0, always prepend a sign.

- [width [*dec-digits*] [shift [*overflow-char
  - [*pad-char*]]] [0] F]
- Fixed-Format Floating-Point. With 0, always prepend a sign.

- [width [*int-digits*] [width [*pad-char*]] [i] [0] S]
- Monetary Floating-Point. Print argument as floating-point number with *dec-digits* after decimal point and *exp-digits* in the signed exponent. With -G, choose either -E or -F. With 0, always prepend a sign.

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

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

{-P} [-P] [-OP] [-OP]}
- Plural. If argument eql 1 print nothing, otherwise print a; do the same for the previous argument; if argument eql 1 print y, otherwise print iex; do the same for the previous argument, respectively.

- [setf %]
- Newline. Print n newlines.

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

{-t} [-t] [-t]
- Conditional Newline. Print a newline like pprint-newline with argument :linear, :fill, :miser, or :mandatory, respectively.

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

- [setf |]
- Page. Print n page separators.

- [setf -]
- Tilde. Print n tildes.

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

- [i] [0] [prefix [-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 0, on the remaining arguments, which are extracted by pprint-pop. With i, prefix and suffix default to ( and ). When closed by -0>, spaces in body are replaced with conditional newlines.
14.2 Packages

package::symbol  ▷ Exported symbol of package.
package::symbol  ▷ Possibly unexported symbol of package.

((names nick*)
 (documentation string)
 (intern interned-symbol*)
 (use used-package*)
 (import-from pkg imported-symbol*)
 (shadowing-import-from pkg std-symbol*)
 (shadow std-symbol*)
 (export exported-symbol*)
 (size int))
▷ Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and std-symbols. Add std-symbols to foo's shadowing list.

(make-package foo *(names (nick*)))
▷ Create package foo.

(rename-package package new-name *(new-names))
▷ Rename package. Return renamed package.

(make-package foo)
▷ Make package foo current.

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

(package-use-list package)

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

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

# Common Lisp use
▷ The current package.

(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)
fn [package ]
▷ 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)
▷ 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)
▷ Make symbols of package shadow any otherwise accessible, equally named symbols from other packages. Return T.

(make-concatenated-stream input-stream*)

(make-broadcast-stream output-stream*)

(make-two-way-stream input-stream-part output-stream-part)

(make-echo-stream from-input-stream to-output-stream)

(make-synonym-stream variable-bound-to-stream)
▷ Return stream of indicated type.

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

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

(concatenated-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)

(two-way-stream-output-stream two-way-stream)

(export-stream-input-stream echo-stream)

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

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

(get-output-stream-string string)
▷ 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.

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

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

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

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

(with-input-from-string (foo string (index index) (decl decl*) (start start) (end end))
▷ Evaluate forms with foo locally bound to input string-stream from string. Return values of forms; store next reading position into index.
Pathnames and Files

Pathname component.

Host pathname conversion.

Pathname conversion.

Default pathname conversion.

User's home directory.

Enough-namestring path-\(or\)-stream

Namestring path-\(or\)-stream

File-namestring path-\(or\)-stream

Directory-namestring path-\(or\)-stream

Host-namestring path-\(or\)-stream

Translate pathname path-\(or\)-stream wildcard-path-\(a\) wildcard-path-\(b\)

Logical pathname logical-\(path\)-\(or\)-stream

Logical pathname translations logical-host

Load-logical pathname translations logical-host

Translate-logical pathname path-\(or\)-stream

Probe file file

Truename file

Write date file

File author file

File length stream

Rename file file

Delete file file

Directory path

Ensure directories exist path [verbose bool]

Packages and Symbols

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

14.1 Predicates

Symbol file

Package file

Keyword file
### 15.4 Declarations

| (proclaim decl) | Globally make declaration(s) decl. decl can be: declaration, type, ftype, inline, notinline, optimize, or special. See below. |
| (declare decl*) | Inside certain forms, locally make declarations decl*. decl can be: dynamic-extent, type, ftype, ignorable, ignore, inline, notinline, optimize, or special. See below. |
| (declaration foo*) | Make foos names of declarations. |
| (dynamic-extent variable* (function function]*) | Declare lifetime of variables and/or functions to end when control leaves enclosing block. |
| (type type variable*) (ftype type function*) | Declare variables or functions to be of type. |
| (ignorable) (ignore (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. |

### 14.3 Symbols

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

| (make-symbol name) | Make fresh, uninterned symbol name. |
| (gensym [\#\$\%\&\^\_\~]) | Return fresh, uninterned symbol \#$$\%&^_~ with n from \#\$\%\&^_~. Increment \#\$\%\&^_~. |
| (copy-symbol symbol [props]) | Return uninterned copy of symbol. If props is T, give copy the same value, function and property list. |
| (symbol-name symbol) | Name, package, property list, value, or function, respectively, of symbol. setfable. |

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

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

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

(defun do-symbols (var [result])
  (declare decl*)
  "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.

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

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

(defun provide module)
  "If not already there, add module to \*modules. Deprecated.

\*modules
  "List of names of loaded modules.
```
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 (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, or T in case of warnings or errors excluding style-warnings.

(/compile-file file ([:output-file out-path
  [:verbose book | :*verbose:*]
  [:print book | :*print:*]
  [:external-format file-format | default])
▷ 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 book | :*verbose:*]
  [:print book | :*print:*]
  [:if-does-not-exist book]
  [:external-format file-format | default])
▷ Load source file or compiled file into Lisp environment. Return T if successful.

(/compile-file) ▷ pathnames | true
▷ Input file used by /compile-file/by /load.

(/compile) ▷ ([print* | :*print:*]
  [:verbose* | :*verbose:*]
▷ Defaults used by /compile-file/by /load.
current from 21:00, ignoring leap seconds.

String representing physical location of computer.

time, respectively, in clock

machine

(name or version of implementation, operating system, or hardware, respectively.)

(machine-instance) ▶ Computer name.

16 External Environment

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

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

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

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

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

(lisp-implementation
(software
(machine
(compile-speed
(compile-speed
(optimize
(safety)

(special var*) ▶ Declare ears to be dynamic.
USER-HOMEDIR-PATHNAME 42
USING 24
V 40
VALUES 18, 33
VALUES-LIST 18
VARIABLE 46
VECTOR 12, 32
VECTOR-PDP 12
VECTOR-PUSH 12
VECTOR-PUSH-EXTEND 12
VECTORP 11

WARN 29
WARNING 32
WHEN 21, 24
WHILE 24
WILD-PATHNAME-P 33
WITH 22
WITH-ACCESSORS 25
WITH-COMPILATION-UNIT 47
WITH-CONDITION-RESTARTS 30
WITH-HASH-TABLE-ITERATOR 15
WITH-INPUT-

FROM-STRING 41
WITH-OPEN-FILE 41
WITH-OPEN-STREAM 41
WITH-OUTPUT-TO-STRING 42
WITH-PACKAGE-ITERATOR 45
WITH-Simple-RESTART 30
WITH-SLOTS 26
WITH-STANDARD-IO-SYNTAX 33
WRITE 37

WRITE-BYTE 36
WRITE-CHAR 36
WRITE-SEQUENCE 36
WRITE-STRING 36
WRITE-TO-STRING 37

Y-OR-N-P 33
YES-OR-NO-P 33

ZEROP 3