Contents

1 Numbers 3
  1.1 Predicates . . . . 3
  1.2 Numeric Functns . 3
  1.3 Logic Functions . 5
  1.4 Integer Functions . 6
  1.5 Implementation-Dependent . . . . 6
2 Characters 7
3 Strings 8
4 Conses 8
  4.1 Predicates . . . . 8
  4.2 Lists . . . . . . 9
  4.3 Association Lists . 10
  4.4 Trees . . . . . . 10
  4.5 Sets . . . . . . 11
5 Arrays 11
  5.1 Predicates . . . . 11
  5.2 Array Functions . 11
  5.3 Vector Functions . 12
6 Sequences 12
  6.1 Seq. Predicates . 12
  6.2 Seq. Functions . 13
7 Hash Tables 15
8 Structures 16
9 Control Structure 16
  9.1 Predicates . . . . 16
  9.2 Variables . . . . 17
  9.3 Functions . . . . 18
  9.4 Macros . . . . . 19
10 CLOS 25
  10.1 Classes . . . . . 25
  10.2 Generic Functns . 26
  10.3 Method Combi-
     nation Types . . . . . 28
11 Conditions and Errors 28
12 Types and Classes 31
13 Input/Output 33
  13.1 Predicates . . . . 33
  13.2 Reader . . . . . 33
  13.3 Character Syntax . 35
  13.4 Printer . . . . . 36
  13.5 Format . . . . . 38
  13.6 Streams . . . . . 40
  13.7 Paths and Files . . . 42
14 Packages and Symbols 43
  14.1 Predicates . . . . 43
  14.2 Packages . . . . 44
  14.3 Symbols . . . . 45
  14.4 Std Packages . . . 46
15 Compiler 46
  15.1 Predicates . . . . 46
  15.2 Compilation . . . 46
  15.3 REPL & Debug . . 47
  15.4 Declarations . . . 48
16 External Environment 49

Typographic Conventions

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; bar]
▷ 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 ^progn; see page 21.

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

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

1.1 Predicates

\[(\text{/= number}^+)\]  \(\triangleright\) T if all numbers, or none, respectively, are equal in value.

\[(\text{/>= number}^+)\]  \(\triangleright\) Return T if numbers are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

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

\[(\text{numberp} \text{foo})\]  \(\triangleright\) T if \(\text{foo}\) is of indicated type.

\[(\text{1+ a})\]  \(\triangleright\) Return \(a + 1\) or \(a - 1\), respectively.

\[(\text{1- a})\]  \(\triangleright\) Increment or decrement the value of \(\text{place}\) by \(\text{delta}\). Return new value.

\[(\text{exp p})\]  \(\triangleright\) Return \(e^p\) or \(b^p\), respectively.

\[(\text{log a [b]])\]  \(\triangleright\) Return \(\log_a\) or, without \(b\), \(\ln\) \(a\).

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

\[(\text{lcm integer}^+)\]  \(\triangleright\) Least common multiple or greatest common denominator, respectively, of integers. (gcd) returns 0.

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

\[(\text{sin a})\]  \(\triangleright\) sin \(a\), cos \(a\), or tan \(a\), respectively. (\(a\) in radians.)

\[(\text{atan a})\]  \(\triangleright\) arctan \(a\) or arccos \(a\), respectively, in radians.
\( \sinh a \), \( \cosh a \), or \( \tanh a \), respectively.

\( \operatorname{asinh} a \), \( \operatorname{acosh} a \), or \( \operatorname{atanh} a \), respectively.

\( e^{ia} = \cos a + i \sin a \).

Return complex conjugate of \( a \).

Greatest or least, respectively, of \( \text{num} \)s.

Return as integer or float, respectively, \( \frac{n}{d} \) rounded, or rounded towards \(-\infty\), \(+\infty\), or 0, respectively; and remainder.

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

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

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

Current random state.

\( \text{num} \cdot a \) with \( \text{num} \cdot a \)’s sign.

Number of magnitude 1 representing sign or phase of \( n \).

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

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

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

Make a complex number.

Angle of \( \text{num} \)’s polar representation.

Return \( |n| \).

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

Convert \( \text{real} \) into \( \text{float} \) with type of \( \text{prototype} \).
1.3 Logic Functions

Negative integers are used in two's complement representation.

\[ (\text{boole} \ \text{operation} \ \text{int}-a \ \text{int}-b) \]
\[ \Rightarrow \text{Return value of bitwise logical operation. operations are} \]
\[ \begin{align*}
\text{boole-1} & \Rightarrow \text{int}-a \\
\text{boole-2} & \Rightarrow \text{int}-b \\
\text{boole-2c} & \Rightarrow \text{~int}-a \\
\text{boole-nand} & \Rightarrow \text{~int}-a \ \text{~int}-b \\
\end{align*} \]

\[ (\text{lognot} \ \text{integer}) \]
\[ \Rightarrow \text{~integer}. \]

\[ (\text{logeqv} \ \text{integer}^*) \]
\[ \Rightarrow \text{Return value of exclusive-nor of anded integers, respectively. Without any integer, return -1}. \]

\[ (\text{logand} \ \text{integer}^*) \]
\[ \Rightarrow \text{Return value of exclusive-ored or ored integers, respectively. Without any integer, return 0}. \]

\[ (\text{logorcl} \ \text{integer} \ \text{integer}) \]
\[ \Rightarrow \text{~int}-a \ \text{~int}-b \]

\[ (\text{logorcl2} \ \text{integer} \ \text{integer}) \]
\[ \Rightarrow \text{~int}-a \ \text{~int}-b \]

\[ (\text{logandn} \ \text{integer} \ \text{integer}) \]
\[ \Rightarrow \text{(int}-a \ \text{int}-b). \]

\[ (\text{logxor} \ \text{integer}^*) \]
\[ \Rightarrow \text{Return value of exclusive-ored or ored integers, respectively. Without any integer, return 0}. \]

\[ (\text{logbitp} \ i \ \text{int}) \]
\[ \Rightarrow \text{T if zero-indexed ith bit of int is set}. \]

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

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

\(\text{\#integer-length integer}\)

- Number of bits necessary to represent integer.

\(\text{\#ld-test byte-spec integer}\)

- Return \(T\) if any bit specified by \(\text{byte-spec}\) in \(\text{integer}\) is set.

\(\text{\#ash integer count}\)

- Return copy of \(\text{integer}\) arithmetically shifted left by \(\text{count}\) adding zeros at the right, or, for \(\text{count} < 0\), shifted right discarding bits.

\(\text{\#byte-spec integer}\)

- Extract \(\text{byte}\) denoted by \(\text{byte-spec}\) from \(\text{integer}\). \(\text{setf}\)able.

\(\text{\#deposit-field\{\text{dpb}\}\space{\text{int-a\space{byte-spec\space{int-b}}}\}}\)

- Return \(\text{int-b}\) with \(\text{byte-spec}\) replaced by corresponding bits of \(\text{int-a}\), or by the low \((\text{byte-size\space{byte-spec}})\) bits of \(\text{int-a}\), respectively.

\(\text{\#mask-field byte-spec integer}\)

- Return copy of \(\text{integer}\) with all bits unset but those denoted by \(\text{byte-spec}\). \(\text{setf}\)able.

\(\text{\#byte size position\space{byte-size byte-spec}}\)

- Byte specifier for a \(\text{byte-size}\) of \(\text{bit}\)s starting at a weight of \(\text{position}\).

\(\text{\#byte-position byte-spec}\)

- Size or \(\text{position}\), respectively, of \(\text{byte-spec}\).

1.5 Implementation-Dependent

\(\text{\#short-\text{float} single-\text{float} double-\text{float} long-\text{float}}\)

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

\(\text{\#least-negative least-negative-normalized least-positive least-positive-normalized}}\)

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

\(\text{\#most-negative most-positive}}\)

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

\(\text{\#decode-float n}}\)

- Return significand, exponent, and sign of \(\text{float}\) \(n\).

\(\text{\#scale-float n [i]}\)

- With \(n\)'s radix \(b\), return \(n^b\).

\(\text{\#float-radix n}}\)

- \(\text{float-digits n}}\)

- \(\text{float-precision n}}\)

- Radix, digits of number in that radix, or precision in that radix, respectively, of \(\text{float}\) \(n\).

\(\text{\#upgraded-complex-part-type \text{foo}}\)

- Type of most specialized \text{complex} number able to hold parts of \text{type \text{foo}}.
3 Strings

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

(strings foo) \(\triangleright\) T if foo is of indicated type.

(simple-string-p foo) \(\triangleright\) Return T if foo is of indicated type.

{string= foo bar \(\triangleright\) \{start1 start-foo \} \{end1 end-foo \}

\(\triangleright\) 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.

(string /= not-equal) \{start1 start-foo \} \{end1 end-foo \}

\(\triangleright\) Return T if subsequences of foo and bar are equal. Obey/ignore, respectively, case.

(string >= greaterp) \(\triangleright\) Return T if foo is greater than bar.

(string >= not-greaterp) \(\triangleright\) Return T if foo is less than bar.

(string < lessp) \(\triangleright\) Return T if foo is less than bar.

(string <= not-lessp) \(\triangleright\) Return T if foo is not less than bar.

(string\(\leq\) not-greaterp) \(\triangleright\) Return T if foo is not greater than bar.

(make-string size \(\{\text{initial-element char} \}\) \(\{\text{element-type type} \}\) \(\{\text{character} \}\)) \(\triangleright\) Return string of length size.

(make-string \(\{\text{size} \}\)) \(\triangleright\) Return string of size.

(string x) \(\triangleright\) Return string of size.

(string-capitalize x) \(\triangleright\) Return string of size.

(string-upcase x) \(\triangleright\) Return string of size.

(string-downcase x) \(\triangleright\) Return string of size.

(nstring-capitalize string) \(\triangleright\) Return string of size.

(nstring-upcase string) \(\triangleright\) Return string of size.

(nstring-downcase string) \(\triangleright\) Return string of size.

(Convert string into a string with capitalized words, an all-upper case string, or an all-lower case string, respectively.

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

(char string i) \(\triangleright\) Return zero-indexed i\text{th} character of string ignoring/obeying, respectively, fill pointer. setable.

(schar string i) \(\triangleright\) Return zero-indexed i\text{th} character of string ignoring/obeying, respectively, fill pointer. setable.

(parse-integer string \(\{\text{start} \}\) \(\{\text{end} \}\) \(\{\text{radix} \}\) \(\{\text{infix} \}\) \(\{\text{junk-allowed} \}\) \(\{\text{bool} \}\)) \(\triangleright\) Return integer parsed from string and index of parse end.

4 Conses

4.1 Predicates

(consp foo) \(\triangleright\) Return T if foo is of indicated type.

(listp foo) \(\triangleright\) Return T if foo is a list.

(endp list) \(\triangleright\) Return T if list is NIL.

(null foo) \(\triangleright\) Return T if foo is NIL.
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 foons 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∗)  ▶ Declare variables or functions to be of type.

(ignite ignore ) (function function)∗)  ▶ Suppress warnings about used/unused bindings.

(inline function∗)  ▶ Tell compiler to integrate/not to integrate, respectively, called functions into the calling routine.
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)
\[
\begin{align*}
&\text{function}\{test\ test-value\ (\text{test not test})
\end{align*}
\]
▷ Return first list after successively applying function to corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

(mapcar function list+)
\[
\begin{align*}
&\text{function}\{\text{return}\{\text{values}\}
\end{align*}
\]
▷ Return list of return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list.

(mapcan function list+)
\[
\begin{align*}
&\text{function}\{\text{values}\}
\end{align*}
\]
▷ Return list of concatenated return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

(mapcar function list+)
\[
\begin{align*}
&\text{function}\{\text{values}\}
\end{align*}
\]
▷ Return list of concatenated return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

(mapalist function list+)
\[
\begin{align*}
&\text{function}\{\text{values}\}
\end{align*}
\]
▷ Return list of concatenated return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

(maplist function list+)
\[
\begin{align*}
&\text{function}\{\text{values}\}
\end{align*}
\]
▷ Return list of concatenated return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

(map function list+)
\[
\begin{align*}
&\text{function}\{\text{values}\}
\end{align*}
\]
▷ Return list of concatenated return values of function successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. function should return a list.

(macroexpand-1 form)
\[
\begin{align*}
&\text{form}\{\text{values}\}
\end{align*}
\]
▷ Return the first form in the macro expansion of form and its arguments.

(macroexpand form)
\[
\begin{align*}
&\text{form}\{\text{values}\}
\end{align*}
\]
▷ Return the entire macro expansion of form and its arguments.

4.4 Trees

(tree-equal foo bar)
\[
\begin{align*}
&\text{test}\ test-value\ (\text{test not test})
\end{align*}
\]
▷ Return T if trees foo and bar have same shape and leaves satisfying test.

(subst new old tree)
\[
\begin{align*}
&\text{test}\ test-value\ (\text{test not function})
\end{align*}
\]
▷ Make copy of tree with each subtree or leaf matching old replaced by new.

(subst-if-not new test tree)
\[
\begin{align*}
&\text{key function}\n\end{align*}
\]
▷ Make copy of tree with each subtree or leaf satisfying test replaced by new.

15.3 REPL and Debugging

(+ ...+)
▷ Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.

(read file-or-function)
▷ Invoke editor if possible.

(macroexpand-1 form)
\[
\begin{align*}
&\text{form}\{\text{values}\}
\end{align*}
\]
▷ Return the first form in the macro expansion of form and its arguments.

(macroexpand form)
\[
\begin{align*}
&\text{form}\{\text{values}\}
\end{align*}
\]
▷ Return the entire macro expansion of form and its arguments.
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 \{\set 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 file \{\:output-file out-path\} \{\:verbose \{\+\\-\+\\\} \{\:print \{\+\\-\+\\\}\}\} \{\:external-format file-format \{\:default\}\})

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

(\compile-file-pathname file \{\:output-file out-path\} \{\other-keyargs\})

▷ Pathname \compile-file writes to if invoked with the same arguments.

(\load path \{\:verbose \{\+\\-\+\\\}\} \{\:print \{\+\\-\+\\\}\}\})

▷ Load source file or compiled file into Lisp environment. Return T if successful.

5 Arrays

5.1 Predicates

(\array? foo) (\vector? foo)

▷ T if foo is of indicated type.

(\bit-vector-p foo) (\simple-bit-vector-p foo)

▷ T if foo is of indicated type.

5.2 Array Functions

(\make-array dimension-sizes \{\adjustable bool\})

▷ Memory allocation and coordination of array specified.

(\adjust-array array dimension-sizes)

▷ Memory allocation and coordination of array specified.

(\aref array \{\subscripts\})

▷ Return array element pointed to by subscripts. setfable.

(\row-major-aref array i)

▷ Return ith element of array in row-major order. setfable.

(\array-rank array)

▷ Number of dimensions of array.

(\array-displacement array)

▷ Target array and offset.
(\textit{bit} \textit{bit-array} [subscripts])
(\textit{sbbit} simple-bit-array [subscripts])
\hspace{1em}⇒ Return element of bit-array or of simple-bit-array, \texttt{setf}-able.

(\textit{bit-not} \textit{bit-array} \texttt{result-bit-array})
\hspace{1em}⇒ Return result of bitwise negation of \textit{bit-array}. If result-bit-array is \texttt{T}, put result in \textit{bit-array}; if it is \texttt{NIL}, make a new array for result.

\hspace{1em}⇒ Return result of bitwise logical operations (cf. operations of \textit{boole}, page 5) on \textit{bit-array-a} and \textit{bit-array-b}. If result-bit-array is \texttt{T}, put result in \textit{bit-array-a}; if it is \texttt{NIL}, make a new array for result.

\texttt{array-rank-limit} \hspace{1em}⇒ Upper bound of array rank; ≥ 8.

\texttt{array-dimension-limit} \hspace{1em}⇒ Upper bound of an array dimension; ≥ 1024.

\texttt{array-total-size-limit} \hspace{1em}⇒ Upper bound of array size; ≥ 1024.

5.3 Vector Functions

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

(\textit{vector} \textit{foo}*) \hspace{1em}⇒ Return fresh simple vector of foos.

(\textit{svector} \textit{vector} \textit{i}) \hspace{1em}⇒ Element \textit{i} of simple vector. \texttt{setf}-able.

(\textit{vector-push} \textit{foo} \textit{vector})
\hspace{1em}⇒ Return \texttt{NIL} if \textit{vector}'s fill pointer equals size of \textit{vector}. Otherwise replace element of \textit{vector} pointed to by \texttt{fill pointer} with \textit{foo}; then increment fill pointer.

(\textit{vector-push-extend} \textit{foo} \textit{vector} \texttt{[num]})
\hspace{1em}⇒ Replace element of \textit{vector} pointed to by \texttt{fill pointer} with \textit{foo}, then increment fill pointer. Extend \textit{vector}'s size by ≥ \texttt{num} if necessary.

(\textit{vector-pop} \textit{vector})
\hspace{1em}⇒ Return element of \textit{vector} its fillpointer points to after decrementation.

(\textit{fill-pointer} \textit{vector}) \hspace{1em}⇒ Fill pointer of \textit{vector}. \texttt{setf}-able.

6 Sequences

6.1 Sequence Predicates

(\textit{every} \textit{test sequence}*)
\hspace{1em}⇒ Return \texttt{NIL} or \texttt{T}, respectively, as soon as \textit{test} on any set of corresponding elements of \textit{sequences} returns \texttt{NIL}.

(\textit{some} \textit{test sequence}*)
\hspace{1em}⇒ Return value of \textit{test} or \texttt{NIL}, respectively, as soon as \textit{test} on any set of corresponding elements of \textit{sequences} returns non-
\texttt{NIL}. 

(package-shadowing-symbols \textit{package})
\hspace{1em}⇒ List of symbols of \textit{package} that shadow any otherwise accessible, equally named symbols from other packages.

(\textit{export} \textit{symbols} \textit{package} \textit{package}*)
\hspace{1em}⇒ Make symbols external to \textit{package}. Return \texttt{T}.

(\textit{unexport} \textit{symbols} \textit{package} \textit{package}*)
\hspace{1em}⇒ Revert symbols to internal status. Return \texttt{T}.

\texttt{do-symbols} \texttt{do-external-symbols} \texttt{do-all-symbols} \texttt{(var \textit{result})}
\hspace{1em}⇒ Evaluate \texttt{tagbody}-like body with \textit{var} successively bound to every symbol from \textit{package}, to every external symbol from \textit{package}, or to every symbol from all registered packages, respectively. Return values of \textit{result}. Implicitly, the whole form is a \texttt{block} named \texttt{NIL}.

(with-package-iterator \textit{foo packages} \texttt{[internal][external][inherited]})
\hspace{1em}⇒ Return values of \textit{forms}. In \textit{forms}, successive invocations of \texttt{(foo)} return \texttt{T} if a symbol is returned; a symbol from \textit{packages}; accessibility \texttt{(internal, external, or :inherited)}; and the package the symbol belongs to.

(require module \texttt{path})*
\hspace{1em}⇒ If not in \texttt{*modules*}, try \textit{paths} to load \textit{module} from. Signal \texttt{error} if unsuccessful. Deprecated.

(provide module)
\hspace{1em}⇒ If not already there, add \textit{module} to \texttt{*modules*}. Deprecated.

\texttt{*modules*} \hspace{1em}⇒ List of names of loaded modules.

14.3 Symbols

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

(make-symbol \textit{name})
\hspace{1em}⇒ Make fresh, uninterned \texttt{symbol} \textit{name}.

(gensym \texttt{#t})
\hspace{1em}⇒ Return fresh, uninterned symbol \texttt{#isn} with \textit{n} from \texttt{*gensym-counter*}. Increment \texttt{*gensym-counter*}.

(gentemp \texttt{prefix [package package]})
\hspace{1em}⇒ Intern fresh symbol in \textit{package}. Deprecated.

(copy-symbol \textit{symbol} \texttt{propertyp})
\hspace{1em}⇒ Return uninterned copy of \textit{symbol}. If \textit{props} is \texttt{T}, give copy the same value, function and property list.

(symbol-name \textit{symbol})

(symbol-package \textit{symbol})

(symbol-plist \textit{symbol})

(symbol-value \textit{symbol})

(symbol-function \textit{symbol})
\hspace{1em}⇒ Name, \texttt{package}, property list, value, or function, respectively, of \textit{symbol}. \texttt{setf}-able.

/documentation \texttt{[setf documentation] new-doc}
\hspace{1em}⇒ Get/set documentation string of \textit{foo} of given type.
14.2 Packages

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

package:symbol ▷ Exported symbol of package.

package::symbol ▷ Possibly unexported symbol of package.

:(defpackage foo <
   (nicknames 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 { nicknames { nick* } })
▷ Create package foo.

(rename-package package new-name { new-nicknames })
▷ Rename package. Return renamed package.

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

(use-package { other-packages [package ] })
▷ 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.

*package* { 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 { find-symbol }) foo [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 { shadowing-import }) symbols [package]
▷ Make symbols internal to package. Return T. In case of a name conflict signal correctable package-error or shadow the old symbol, respectively.

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

6.2 Sequence Functions

(make-sequence sequence-type size [initial-element foo])
▷ Make sequence of sequence-type with size elements.

(concatenate type sequence*)
▷ Return concatenated sequence of type.

(merge type sequence-a sequence-b test [key function])
▷ Return interleaved sequence of type. Merged sequence will be sorted if both sequence-a and sequence-b are sorted.

(fill sequence foo { :start start } :end end)
▷ Return sequence after setting elements between start and end to foo.

(length sequence)
▷ Return length of sequence (being value of fill pointer if applicable).

(count sequence)
▷ Return number of elements in sequence which match foo.

(count-if test sequence)
▷ Return number of elements in sequence which satisfy test.

(elt sequence index)
▷ Return element of sequence pointed to by zero-indexed index. setfable.

(subseq sequence start [end])
▷ Return subsequence of sequence between start and end. setfable.

(sort sequence test [key function])
▷ Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

(reverse sequence)
▷ Return sequence in reverse order.

(find-position foo sequence)
▷ Return first element in sequence which matches foo, or its position relative to the begin of sequence, respectively.
Common Lisp Quick Reference

- **find-if**
  - **find-if-not**
  - **position-if**
  - **position-if-not**
  
  ▷ Return first element in `sequence` which satisfies `test`, or its position relative to the begin of `sequence`, respectively.

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

- **remove**
  - `foo` `sequence`

- **delete**
  - `foo` `sequence`

- **remove-if**
  - `remove-if-not`
  - `delete-if`
  - `delete-if-not`
  
  ▷ Make copy of `sequence` without elements matching `foo`.

- **remove-duplicates** `sequence`

- **delete-duplicates** `sequence`

- **substitute**
  - `new` `old sequence`

- **nsubstitute**
  - `new` `old sequence`

- **substitute-if**
  - `substitute-if-not`
  
  ▷ Make copy of `sequence` with all (or count) `olds` replaced by `new`.

- **replace**
  - `sequence-a` `sequence-b`

- **map**
  - `type` `function` `sequence`

- **enough-namestring**
  - `path-or-stream`

- **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`

- **translate-pathname**

- **pathname** `path-or-stream`

  ▷ Pathname of `path-or-stream`.

- **logical-namestring**
  - `logical-path-or-stream`

  ▷ Logical pathname of `logical-path-or-stream`. Logical pathnames are represented as all-uppercase "[host:][]{[dir][***][name]*}{[type]*}", where  

  ```
  ...[
    ...{[version]*[newest]:[newest]}
  ...
  ```

- **logical-namestring-translations**

- **load-logical-namestring-translations**

  ▷ List of `(from-wildcard to-wildcard)` translations for `logical-host`. Settable.

- **translate-logical-namepath** `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`

- **directory** `path`

  ▷ List of pathnames matching `path`.

- **ensure-directories-exist**
  - `path` `{[verbose bool]}

  ▷ Create parts of `path` if necessary. Second return value is `T` if something has been created.

14 Packages and Symbols

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

14.1 Predicates

- **symbolp** `foo`

- **packagep** `foo`

  ▷ `T` if `foo` is of indicated type.

- **keywordp** `foo`
13.7 Pathnames and Files

(/make-pathname
  :host {host NIL unspecified}
  :device {device NIL unspecified}
  (directory {absolute} {relative} {wild} {wild-inferiors} {up} {back} {local} {common} {from-end} {start} {from-end} {start}
  :name {file-name} {wild} {wild-inferiors} {up} {back}
  :type {file-type} {wild} {wild-inferiors} {up} {back}
  :version {newest} {version} {wild} {wild-inferiors} {up} {back}
  :case {local} {common} {local}
  :pathname-host
  :pathname-device
  :pathname-directory
  :pathname-name
  :pathname-type
  :pathname-version path-or-stream (case {local} {common} {local}))
  path-or-stream
  :case {local} {common} {local})
  path-or-stream
  :case {local} {common} {local})

(/parse-namestring foo [host [default-pathname defpath]
  {start start} {end end} [junk-allowed bool])
  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)
  [default-pathname-defaults]
  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-homedirpathname [host])
  return user’s home directory.

(/map-into result-sequence function sequence*)
  return result-sequence with values of function applied to corresponding elements of the sequences.

(/reduce function sequence
  :start start
  :end end
  :key function
  starting with the first two elements of sequence, apply function successively to its last return value together with the next element of sequence. Return last value of function.

(/copy-seq sequence)
  copy of sequence with shared elements.

7 Hash Tables

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

(/hash-table-p foo)
  return T if foo is of type hash-table.

(/make-hash-table
  :size int
  :hash-table-rehash-threshold num
  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. settable.

(/hash-table-count hash-table)
  return 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.

(/rhash hash-table)
  return 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 (existent) *form*)
  return values of forms. In forms, invocations of (foo) return: T if an entry is returned; its key; its value.

(/hash-table-test hash-table)
  test function used in hash-table.

(/hash-table-size hash-table)

(/hash-table-rehash-size hash-table)

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

(/sxhash foo)
  hash code unique for any argument ,equal foo.
8 Structures

(defstruct foo
  (:conc-name [slot-prefix foo])
  :constructor (make-foo [ord-\lambda])
  :copier (copy-\lambda)
  (:include struct (slot [init] [:type sl-type :read-only \lambda]))
  :type list vector
  :vector type
  :print-object [op-printer]
  :print-function [f-printer]
  :predicate [print-\lambda]
  (:predict \lambda)
  (:doc (slot [init] [:type sl-type :read-only bool]))
)

> Define structure foo together with functions make-foo, copy-foo and foo-P; and set\tableable accessor foo-slot. Instances 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-\lambda (see page 18) is given, by \{maker \* {key value\*}\}. In the latter case, args and :keys correspond to the positional and keyword parameters defined in ord-\lambda whose var's in turn correspond to slots.

:print-object/:print-function generate a :print-object method for an instance bar of foo calling (o-printer bar stream) 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)  \> \T if foo and bar are identical.

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

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

(equalp foo bar)
> \T if foo and bar are identical; or are the same character ignoring case; or are numbers of the same value ignoring type; or are equivalent pathnames; or are conses or arrays of the same shape with 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)  \> \T if foo is NIL; NIL otherwise.

(boundp symbol)  \> \T if symbol is a special variable.
(13.6 Streams)

```
(open path
  :direction input
  :output io
  :probe nil
  :element-type
    :type symbol
    :default nil
  :new-version
    :error nil
    :rename nil
    :rename-and-delete nil
  :if-exists
    :overwrite nil
    :append nil
    :supersede nil
  :if-does-not-exist
    :error nil
    :create nil
  :external-format
    :name form
    :default form)

▷ Open file-stream to path.
```

9.2 Variables

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

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

(defvar foo [form [doc]])
  ▷ Unbound list, assign of 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.

(setf 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.

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

(rotatem place*)
  ▷ Rotate values of places left, old first becoming new last place’s value. Return NIL.

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

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

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

(remprop symbol key)

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

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

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

(let* {name (name [value])} (declare decl* form)
```

(\constantp foo [environment] t)
  ▷ \t if foo is a constant form.

(\functionp foo)
  ▷ \t if foo is of type function.

(\boundp {foo (\setf foo)})
  ▷ \t if foo is a global function or macro.
9.3 Functions

Below, ordinary lambda list (\(\text{ord-}\lambda^*)\) has the form

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

\[\text{[var} \text{init} [\text{supplied-p}]]\] \cup & \text{allow-other-keys}]

\[\text{[var} \text{init} [\text{supplied-p}]]\). \]

\[\text{supplied-p is } T \text{ if there is a corresponding argument. } \text{init forms can refer to any } \text{init } \text{and supplied-p to their left.}
\]

\[
\left\{\begin{align*}
& \text{defun} \{\text{foo} (\text{ord-}\lambda^*) (\text{new-value} \text{ord-}\lambda^*) \{\text{declare} \text{dec}^* \{\text{doc} \text{forms}\} \}
\quad \text{Defer function named } \text{foo} \text{ or (setf } \text{foo}) \text{, or an anonymous function, respectively, which applies forms to } \text{ord-}\lambda^* . \text{ For \text{defun forms are enclosed in an implicit } \text{block named } \text{foo}.} \\
& \text{mlambda} \{\text{ord-}\lambda^*) \text{forms}^* \}
\quad \text{Define a function named } \text{foo} \text{ or (setf } \text{foo}) \text{, or an anonymous function, respectively, which applies forms to } \text{ord-}\lambda^* . \text{ For \text{defun forms are enclosed in an implicit } \text{block named } \text{foo}.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{flet} \{\text{labels} \}
\quad \text{Evaluate forms with locally defined functions } \text{foo} \text{. Globally defined functions of the same name are shadowed. Each } \text{foo} \text{ is also the name of an implicit } \text{block} \text{ around its corresponding } \text{local-form}^* . \text{ Only for } \text{labels, functions } \text{foo} \text{ are visible inside } \text{local-forms. Return values of forms.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{function} \{\text{foo} \{\text{mlambda \text{form}^*}\} \}
\quad \text{Defer function named } \text{foo} \text{ or a lexical closure of the } \text{mlambda expression.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{apply} \{\text{function} \{\text{setf function} \}\} \arg^* \text{args}
\quad \text{Values of function called with } \text{args} \text{ and the list elements of } \text{args. } \text{setfable if function is one of } \text{aref, bit, and } \text{sbit.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{funcall} \{\text{function} \arg^*\}
\quad \text{Values of function called with } \text{args.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{multiple-value-call} \{\text{function} \arg^*\}
\quad \text{Call function with all the values of each } \text{form} \text{ as its arguments. Return values returned by function.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{values-list} \{\text{list}\}
\quad \text{Return elements of } \text{list.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{values} \{\text{foo}\}
\quad \text{Return as multiple values the } \text{primary values of the } \text{foo}. \text{setfable.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{multiple-value-list} \{\text{form}\}
\quad \text{List of the values of } \text{form}.
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{nth-value} n \{\text{form}\}
\quad \text{Zero-indexed } n \text{th return value of } \text{form.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{complement} \{\text{function}\}
\quad \text{Return new function with same arguments and same side effects as } \text{function} \text{, but with complementary truth value.}
\end{align*}\right.
\]

\[
\{\text{R} \{\text{R} \{\text{OR} \{\text{R}\}\}\}\}
\quad \text{Roman. Take argument as number and print it as } \text{English cardinal number, as } \text{English ordinal number, as } \text{Roman numeral, or as old Roman numeral, respectively.}
\]

\[
\left\{\begin{align*}
& \text{width} \{\text{[i] \text{pad-char} \{\text{comma-} \text{inter-val} \{\text{rule}\}\}\} [i] \{\text{0} \{\text{0} \{\text{0} \{\text{0}\}\}\}\}\}
\quad \text{Decimal/Binary/Octal/Hexadecimal. Print integer argument as number. With } \text{i, } \text{group digits comma-interval each; with } \text{0, } \text{always prepend a sign.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{dec-digits} \{\text{[int-digits} \{\text{overflow-char}\} \{\text{pad-char} \{\text{exp-char}\}\} \} \}
\quad \text{Fixed-Format Floating-Point. With } \text{0, } \text{always prepend a sign.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{dec-digits} \{\text{[exp-digits} \{\text{scale-factor}\} \{\text{overflow-char}\} \{\text{exp-char}\}\} \}
\quad \text{Exponential/General Floating-Point. Print argument as floating-point number with } \text{dec-digits after decimal point and } \text{exp-digits in the signed exponent. With } \text{0-G, choose either } \text{-E} \text{ or } \text{-F}. \text{With } \text{0, } \text{always prepend a sign.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{[C}-\text{C] [-0-C] [-C]} \{\text{E} \{\text{G}\}\}
\quad \text{Character. Print, spell out, print in } \#\ \text{syntax, or tell how to type, respectively, argument as (possibly non-printing) character.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{text} \{\text{text} \{\text{text} \{\text{-text}\}\}\}\{\text{-text}\}
\quad \text{Case-Conversion. Convert text to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{[P}-\text{P] [-0-P] [-P]} \{\text{P}\}
\quad \text{Plural. If argument } \text{eq1 print nothing, otherwise print } \text{a; do the same for the previous argument; if argument } \text{eq1 print y, otherwise print } \text{ies; do the same for the previous argument, respectively.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{[C]-[C]} \{\text{P}-\text{P}\} \{\text{C}\}
\quad \text{Conditional Newline. Print a newline like } \text{print-newline with argument } \text{linear, } \text{fill, } \text{miser, or } \text{mandatory, respectively.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \text{[C]-[C]} \{\text{C}\}
\quad \text{Ignored Newline. Ignore newline, or whitespace following newline, or both, respectively.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \{\text{R} \} \{\text{R}\}
\quad \text{Page. Print n page separators.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \{\text{W} \} \{\text{W}\}
\quad \text{Tilde. Print n tildes.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \{\text{min-col} \{\text{[col-in-cel]} \{\text{[min-pad} \{\text{pad-char} \{\text{char}\}\}\]\{\text{width}\}\}\}{\text{text} \{\text{text} \} \{\text{text} \}\}\text{text} \rightarrow \text{text}
\quad \text{Justification. Justify text produced by texts in a field of at least min-col columns. With } \text{t, right justify; with } \text{0, left justify. If this would leave less than spare characters on the current line, output nl-text first.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \{\text{C} \}
\quad \text{Logical Block. Act like } \text{print-print-logical-block using body as } \text{format control string on the elements of the list argument or, with } \text{0, on the remaining arguments, which are extracted by } \text{print-pop. With } \text{t, prefix and suffix default to ( and )}. \text{ When closed by } \text{-0-}, \text{ spaces in body are replaced with conditional newlines.}
\end{align*}\right.
\]

\[
\left\{\begin{align*}
& \{\text{R} \{\text{R} \{\text{OR} \{\text{R}\}\}\}\}
\quad \text{Roman. Take argument as number and print it as } \text{English cardinal number, as } \text{English ordinal number, as } \text{Roman numeral, or as old Roman numeral, respectively.}
\end{align*}\right.
\]
(set-print-dispatch type function [priority E] {table [print-print-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.

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

(copy-print-dispatch [table [print-print-dispatch]])
  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 control)
  Return function of stream and arg* applying format to stream, control, and arg* returning NIL or any excess args.

(format [T]NIL [out-string] out-string) control arg*)
  Output string control which may contain * directives possibly taking any args. Alternatively, control can be a function returned by formatter which is then applied to out-string and arg*. Output to out-string, out-string 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]]
  Aesthetic/Standard. Print argument of any type for consumption by humans/readers, respectively. With ;, print NIL as () rather than NIL; with @, add pads-chars on the left rather than the right.

- [radius] [width] [pad-char] [comma-char]
  Radix. (With one or more prefix arguments.) Print argument as number; with @, group digits comma-prefix each; with @, always preprend a sign.
(symbol-macrolet ((foo expansion-form*) (declare decl*) form*)
  \> 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
  (var* [optional] var [init [supplied-p]])
  [rest var] [key {var [(key var) [init [supplied-p]]]]}
  [allow-other-keys [environment var]]
  \> Specify how to set a place accessed by function.

Short form (setf (function arg*) value-form) is replaced by
(setf 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* describes 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 `block`
named function.

(define-set-expander function (macro-lambda*) (declare decl*) [doc]
  form*)
  \> Specify how to set a place accessed by function. On

Inavocation 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 get-set-expansion
where the elements of macro lambda list macro-lambda
are bound to corresponding args. forms are enclosed in an implicit
`block` named function.

(get-set-expansion place [environment])
  \> 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 set and how to read place.

(define-modify-macro foo ([optional]
  var [init [supplied-p]]) [rest var]) function [doc]
  \> 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
  \> List of macro lambda list keywords. These are at least:

  \&whole var
  \> Bind var to the entire macro call form.

  \&optional var*
  \> Bind vars to corresponding arguments if any.

  \&rest \&body var
  \> Bind var to a list of remaining arguments.

  \&key var*
  \> Bind vars to corresponding keyword arguments.

  \&allow-other-keys
  \> Suppress keyword argument checking. Callers can do

  so using `allow-other-keys T`

  \&environment var
  \> Bind var to the lexical compilation environment.

  \&aux var*
  \> Bind vars as in `let`.

  \> 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`).

(setstream keyword with `write only`.

(pprint-fill stream foo [parenthesis [noop]])
(pprint-tabular stream foo [parenthesis [noop [nsp]]])
(pprint-linear stream foo [parenthesis [noop]])

\> Print foo to stream. If foo is a list, print as many elements

per line as possible; do the same in a table with a column
width of n ems; or print either all elements on one line or
each on its own line, respectively. Return NIL. usable with
`format directive -/.

(pprint-logical-block stream list {prefix string per-line-prefix string
suffix string})
(declare decl*) form*)
  \> Evaluate 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.

(pprint-pop)
  \> Take next element off list. If there is no remaining tail of

list, or `*print-length*` or `*print-circles` indicate printing
should end, send element together with an appropriate
indicator to stream.

(pprint-tab [line [line-relative] [section [section-relative]]]
  c i [stream standard-output])
  \> Move cursor forward to column number c + ki, k \geq 0

being as small as possible.

(pprint-indent [block [current] n stream standard-output])
  \> Specify indentation for innermost logical block relative to

leftmost position to current position. return NIL

otherwise.

(pprint-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 is a pretty printing

stream. return NIL.

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

\*print-base*
  \> Radix for printing rationals, from 2 to 36.
9.5 Control Flow

(if test then (else))
  ▷ Return values of then if test returns T; return values of else otherwise.

(mcond (test then *)
  ▷ Return the values of the first then* whose test returns T;
     return NIL if all tests return NIL.

(mwhen (test foo *)
  ▷ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.

(mcase test (key) [otherwise] bar *)
  ▷ Return the values of the first foo* one of whose keys is eql test.
      Return values of form if there is no matching key.

(mcase test (key) foo *)
  ▷ Return the values of the first foo* one of whose keys is eql test.
      Signal non-correctable/correctable type-error if there is no matching key.

(mand form *)
  ▷ Evaluate forms from left to right. Immediately return NIL
      if one form’s value is NIL. Return values of last form otherwise.

(mor form *)
  ▷ Evaluate forms from left to right. Immediately return
      the primary value of first non-NIL-evaluating form, or all values
      if last form is reached. Return NIL if no form returns T.

(progn form *)
  ▷ Evaluate forms sequentially. Return values of last form.

(multiple-value-prog1 form-r form *)
(multiple-value-prog2 form-r form *)
  ▷ Evaluate forms in order. Return values/primary value, respectively, of form-r.

(declare (declare decl*) (tag form *)
  ▷ Evaluate tagbody-like body with names lexically bound
      (in parallel or sequentially, respectively) to values. Return NIL or explicitly
      returned values. Implicitly, the whole form is a block named NIL.

(unwind-protect protected cleanup *)
  ▷ Evaluate protected and then, no matter how control leaves
      protected, cleanups. Return values of protected.

(block name form *)
  ▷ Evaluate forms in a lexical environment, and return their values
      unless interrupted by return-from.

(return-from foo result *)
(return result *)
  ▷ Have nearest enclosing block named foo/named NIL, respectively,
      return with values of result.

(tagbody (tag form *))
  ▷ Evaluate forms in a lexical environment. tags (symbols or integers) have lexical scope and dynamic extent, and are
      targets for go. Return NIL.

(go tag)
  ▷ Within the innermost possible enclosing tagbody, jump to
      a tag eql tag.
(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
  eq tag return with the values of form.

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

9.6 Iteration

{do
  {{var [start [step]]}
    (setup form)
  }
  {stop result}
  (decal decl)*
} (tag form)
▷ 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.

(odotimes (var i [result m])
  (decla 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.

(odolist (var list [result m])
   (decla 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.

{with [var-s] [var-s]} [d-type] [= foo]+
  {and [var-p] [var-p]} [d-type] [= bar]+
where destructuring type specifier d-type has the form

{fixnum float NIL of-type [type]}*
▷ Initialize (possibly trees of) local variables var-s sequentially and var-p in parallel.

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

{upfrom|from|downfrom} start
▷ Start stepping with start

{upto|down|above|below} form
▷ Specify form as the end value for stepping.

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

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

(set-dispatch-macro-character char sub-char function)
▷ 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 [readable])
▷ 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.

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

^ 
▷ Begin and end of a string.

'foo
▷ (quote foo); foo unevaluated.

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

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

#Bn; #O-n; #X-n; #R-n
▷ Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.

n/d
▷ The ratio n/d.

{m}[n]^[SUDLE][S][S][S][S][S]m^[n][SUDLE]x
▷ m.n·10^x 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.

#nA sequence
▷ n-dimensional array.

#[n](foo^)
▷ Vector of some (or n) foos filled with last foo if necessary.

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

(read-char stream [standard-input (eof-error (eof-val #f))]
  ➢ Return next character from stream.

(read-char-no-hang stream [standard-input (eof-error (eof-val #f))]
  ➢ Next character from stream or #f, if none is available.

(peek-char mode stream [standard-input (eof-error (eof-val #f))]
  ➢ Next, or if mode is T, next non-whitespace character, or if mode is a character, next instance of it, from stream without removing it there.

(unread-char character stream)
  ➢ Put last read character back into stream; return #f.

(read-byte stream (eof-error (eof-val #f)))
  ➢ Read next byte from binary stream.

(read-line stream (eof-error (eof-val #f))
  ➢ Return a line of text from stream and #f if line has been ended by end of file.

(read-sequence sequence stream [start (end (readlist ))]
  ➢ Replace elements of sequence between start and end with elements from binary or character stream. Return index of sequence’s first unmodified element.

(readable-case readable)
  ➢ Case sensitivity attribute (one of :upcase, :downcase, :preserve, :invert) of readable. Settable.

(copy-readable from-readable to-readable)
  ➢ Return copy of from-readable.

(set-syntax-from-char to-char from-char [to-readable readable]
  ➢ Copy syntax of from-char to to-readable. Return #t.

*readable* ➢ Current readable.

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

*read-default-float-format* ➢ Floating point format to use when not indicated in the number read.

*read-suppress* ➢ If #t, reader is syntactically more tolerant.

(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 #t [readtable]
  ➢ Reader macro function associated with char, and #t if char is a non-terminating macro character.

(make-dispatch-macro-character char [non-term-p]
  ➢ Make char a dispatching macro character. Return #t.

Figure 1: Loop Facility, Overview.
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-symbols | present-symbols| external-symbol | external-symbols} [of in]
  ➔ 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 | nconc|ing} {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 nloop after num iterations; num is evaluated once.

while until test
  ➔ Continue iteration until test returns NIL or T, respectively.

(ndeftype foo (macro-λ*) (declare decl*) [doc] form*)
  ➔ Define type foo which when referenced as (foo any*) (or as foo if macro-λ doesn’t contain any required parameters) applies expanded forms to args renaming the new type. For (macro-λ*) see page 19 but with default value of * instead of NIL. forms are enclosed in an implicit block named foo.

(eq 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* #)
  ➔ Type specifier for intersection of types.

(or type* #:)
  ➔ 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

(stream foo)
  ➔ T if foo is of indicated type.

(pathnamep foo)
  ➔ T if foo is of indicated type.

(readtablep foo)
  ➔ T if foo is of indicated type.

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

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

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

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

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

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

_mdefclass foo (superclass

{always|never} test
  ▶ Terminate _mloop_ returning NIL and skipping any finally parts as soon as test is NIL or T, respectively. Otherwise continue _mloop_ with its default return value set to T.

_thereis test_ → Terminate _mloop_ when test is T and return value of test, skipping any finally parts. Otherwise continue _mloop_ with its default return value set to NIL.

_(mloop-finish)_ → Terminate _mloop_ immediately executing any finally clauses and returning any accumulated results.

Figure 2: Precedence Order of System Classes ( ), Classes ( ), Types ( ), and Condition Types ( ). Every type is also a supertype of NIL, the empty type.
(change-class instance new-class {initarg value} other-keyarg)
▷ Change class of instance to new-class. Retain the status of any slots that are common between instance’s original class and new-class. Initialize any newly added slots with the values of the corresponding initargs if any, or with the values of their :initform forms if not.

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

(initialize-instance instance)

(update-instance-for-different-class previous current)
▷ 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 value} other-keyarg)
▷ On behalf of make-instances-obsolete and by means of shared-initialize, set any initarg slots to their corresponding values; set any remaining added-slots to the values of their :initform forms. Not to be called by user.

(allocate-instance class {initarg value} 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 the 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-unbound class instance slot)
▷ 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+)
▷ T if enclosing method has a next method.

(defgeneric (foo setf foo) (required-var+ [optional] var+ [var]) &rest var+ &key [key var] [key var] [allow-other-keys])
▷ Define or modify generic function foo. Remove any methods previously defined by defgeneric. If 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)) )
> Return values of \( \text{forms} \) unless \( \text{restart} \) is called during their evaluation. In this case, describe \( \text{restart} \) using \( \text{format} \) control and \( \text{args} \) (see page 38) and return \( \text{NIL} \).

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

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

\[
\text{declare \( \text{restart} \) \( \text{form} \) \( \text{arg}^* \) \( \text{form}^b \)}
\]

\[
\text{invoke-restart \( \text{arg}^* \)}
\]

\[
\text{invoke-restart-interactively \( \text{restart} \)}
\]

\[
\text{find-restart name} \ [\text{condition}]
\]

\[
\text{restart-name \( \text{restart} \)}
\]

\[
\text{abort \( \text{muffle-warning} \)}
\]

\[
\text{continue \( \text{store-value value} \)}
\]

\[
\text{use-value value} \ [\text{condition}]
\]

\[
\text{with-condition-restarts condition \( \text{restarts} \) \( \text{form}^b \)}
\]

\[
\text{arithmetic-error-operation condition}
\]

\[
\text{arithmetic-error-operands condition}
\]
10.3 Method Combination Types

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

\texttt{append\ list\ conc\ prog\ max\ min\ +}
➤ Simple built-in \texttt{method-combination} types; have the same usage as the \texttt{c-type} defined by the short form of \texttt{define-method-combination}.

\texttt{\(\texttt{define-method-combination} \ c-type\)}
\begin{verbatim}
\begin{Verbatim}
\{\texttt{documentation} \texttt{string}\}
\{\texttt{identity-with-one-argument} \texttt{book} \texttt{list}\}
\end{Verbatim}
\end{verbatim}

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

\texttt{\(\texttt{define-method-combination} \ c-type \ (\texttt{ord-}\lambda\texttt{-}\texttt{\*)})\)}
\begin{verbatim}
\begin{Verbatim}
\{\texttt{group} \texttt{group}\}
\{\texttt{description} \texttt{control}\}
\{\texttt{order} \texttt{most-specific-first} \texttt{most-specific-last} \texttt{most-specific-first} \texttt{most-specific-first}\}
\{\texttt{required} \texttt{bool}\}
\{\texttt{arguments} \texttt{method-combination-\texttt{\#\*)}\}
\{\texttt{generic-function} \texttt{symbol}\}
\{\texttt{body} \texttt{body}\}
\end{Verbatim}
\end{verbatim}

➤ Long Form. Define new \texttt{method-combination} \texttt{c-type}. A call to a generic function using \texttt{c-type} will be equivalent to a call to the forms returned by \texttt{body} with \texttt{ord-\lambda} bound to \texttt{c-type} (cf. \texttt{defgeneric}), with \texttt{symbol} bound to the generic function, with \texttt{method-combination-\texttt{\#\*)} bound to the arguments of the generic function, and with \texttt{groups} bound to lists of methods. An applicable method becomes a member of the leftmost group whose \texttt{predicate} or \texttt{qualifiers} match. Methods can be called via \texttt{call-method}. Lambda lists (\texttt{ord-\lambda}) and (\texttt{method-combination-\texttt{\#\*)}) according to \texttt{ord-\lambda} on page 18, the latter enhanced by an optional \texttt{&whole} argument.

\texttt{\(\texttt{call-method}\)}
\begin{verbatim}
\begin{Verbatim}
\{\texttt{method} \texttt{(\texttt{make-method form})}\}
\{\texttt{next-method} \texttt{(\texttt{make-method form})}\}
\end{Verbatim}
\end{verbatim}

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

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 32.