## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Predicates</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Numeric Functions</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Logic Functions</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Integer Functions</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Implementation-Dependent</td>
<td>6</td>
</tr>
<tr>
<td>Characters</td>
<td>7</td>
</tr>
<tr>
<td>Strings</td>
<td>8</td>
</tr>
<tr>
<td>4 Conses</td>
<td>8</td>
</tr>
<tr>
<td>4.1 Predicates</td>
<td>8</td>
</tr>
<tr>
<td>4.2 Lists</td>
<td>9</td>
</tr>
<tr>
<td>4.3 Association Lists</td>
<td>10</td>
</tr>
<tr>
<td>4.4 Trees</td>
<td>10</td>
</tr>
<tr>
<td>4.5 Sets</td>
<td>11</td>
</tr>
<tr>
<td>Arrays</td>
<td>11</td>
</tr>
<tr>
<td>5.1 Predicates</td>
<td>11</td>
</tr>
<tr>
<td>5.2 Array Functions</td>
<td>11</td>
</tr>
<tr>
<td>5.3 Vector Functions</td>
<td>12</td>
</tr>
<tr>
<td>Sequences</td>
<td>12</td>
</tr>
<tr>
<td>6.1 Seq. Predicates</td>
<td>12</td>
</tr>
<tr>
<td>6.2 Seq. Functions</td>
<td>13</td>
</tr>
<tr>
<td>Hash Tables</td>
<td>15</td>
</tr>
<tr>
<td>7 Packages and Symbols</td>
<td>43</td>
</tr>
<tr>
<td>14.1 Predicates</td>
<td>43</td>
</tr>
<tr>
<td>14.2 Packages</td>
<td>44</td>
</tr>
<tr>
<td>14.3 Symbols</td>
<td>45</td>
</tr>
<tr>
<td>14.4 Std Packages</td>
<td>46</td>
</tr>
<tr>
<td>Structures</td>
<td>16</td>
</tr>
<tr>
<td>9.1 Predicates</td>
<td>16</td>
</tr>
<tr>
<td>9.2 Variables</td>
<td>17</td>
</tr>
<tr>
<td>9.3 Functions</td>
<td>18</td>
</tr>
<tr>
<td>9.4 Macros</td>
<td>19</td>
</tr>
<tr>
<td>Control Structure</td>
<td>16</td>
</tr>
<tr>
<td>10.1 Classes</td>
<td>25</td>
</tr>
<tr>
<td>10.2 Generic Functions</td>
<td>26</td>
</tr>
<tr>
<td>10.3 Method Combination Types</td>
<td>28</td>
</tr>
<tr>
<td>Conditions and Errors</td>
<td>28</td>
</tr>
<tr>
<td>11.1 Classes</td>
<td>31</td>
</tr>
<tr>
<td>11.2 Types and Classes</td>
<td>33</td>
</tr>
<tr>
<td>Input/Output</td>
<td>33</td>
</tr>
<tr>
<td>13.1 Predicates</td>
<td>33</td>
</tr>
<tr>
<td>13.2 Reader</td>
<td>33</td>
</tr>
<tr>
<td>13.3 Character Syntax</td>
<td>35</td>
</tr>
<tr>
<td>13.4 Printer</td>
<td>36</td>
</tr>
<tr>
<td>13.5 Format</td>
<td>38</td>
</tr>
<tr>
<td>13.6 Streams</td>
<td>40</td>
</tr>
<tr>
<td>13.7 Paths and Files</td>
<td>42</td>
</tr>
<tr>
<td>External Environment</td>
<td>49</td>
</tr>
</tbody>
</table>

## Typographic Conventions

- **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}**
  - Either foo, or bar, or baz.

- **{foo; bar; baz}**
  - Anything from none to each of foo, bar, and baz.

- **foo**
  - Argument foo is not evaluated.

- **bar**
  - Argument bar is possibly modified.

- **foo**
  - foo* is evaluated as in `progn`; see page 21.

- **foo; bar; baz**
  - Primary, secondary, and n'th return value.

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

1.1 Predicates

\( (\text{=} \text{number}^+) \)
\( (\neq \text{number}^+) \)
\( \triangleright \top \) if all numbers, or none, respectively, are equal in value.

\( (\geq \text{number}^+) \)
\( (\nleq \text{number}^+) \)
\( \triangleright \top \) if numbers are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

\( (\text{minusp} \ a) \)
\( (\text{zerop} \ a) \)
\( (\text{plusp} \ a) \)
\( \triangleright \top \) if \( a < 0 \), \( a = 0 \), or \( a > 0 \), respectively.

\( (\text{evenp} \ \text{int}) \)
\( (\text{oddp} \ \text{int}) \)
\( \triangleright \top \) if \( \text{int} \) is even or odd, respectively.

\( (\text{numberp} \ \text{foo}) \)
\( (\text{realp} \ \text{foo}) \)
\( (\text{rationalp} \ \text{foo}) \)
\( (\text{floatp} \ \text{foo}) \)
\( (\text{integerp} \ \text{foo}) \)
\( (\text{complexp} \ \text{foo}) \)
\( (\text{random-state-p} \ \text{foo}) \)
\( \triangleright \top \) if \( \text{foo} \) is of indicated type.

1.2 Numeric Functions

\( (+ \ a \ b) \)
\( (\times \ a \ b) \)
\( \triangleright \sum a \) or \( \prod a \), respectively.

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

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

\( \{\text{incf} \ a \} \)
\( \{\text{decf} \ a \} \)
\( \triangleright \text{Increment} \) or decrement the value of \( \text{place} \) by \( \text{delta} \). Return new value.

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

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

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

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

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

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

\( (\text{asin} \ a) \)
\( (\text{acos} \ a) \)
\( (\text{atan} \ a \ [\text{base}]) \)
\( \triangleright \arcsin a \) or \( \arccos a \), respectively, in radians.
\( \text{sinh} \ a \) \( \rightarrow \) \( \sinh a, \cosh a, \text{ or } \tanh a \), respectively.

\( \text{cosh} \ a \) \( \rightarrow \) \( \sinh a, \cosh a, \text{ or } \tanh a \), respectively.

\( \text{tanh} \ a \) \( \rightarrow \) \( \sinh a, \cosh a, \text{ or } \tanh a \), respectively.

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

\( \text{conjugate} \ a \) \( \rightarrow \) Return complex conjugate of \( a \).

\( \text{max} \ num^+ \) \( \rightarrow \) Greatest or least, respectively, of \( num \).

\( \text{min} \ num^+ \) \( \rightarrow \) Return as integer or float, respectively, \( n/d \) rounded, or rounded towards \( -\infty, +\infty, \) or 0, respectively; and remainder.

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

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

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

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

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

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

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

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

\( \text{imagpart number} \) \( \rightarrow \) Make a complex number.

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

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

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

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

Negative integers are used in two's complement representation.

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

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

\[ \text{boole-1} \rightarrow \text{int-a} \]

\[ \text{boole-2} \rightarrow \text{int-b} \]

\[ \text{boole-2} \rightarrow \text{int-a} \]

\[ \text{boole-2c} \rightarrow \text{int-a} \]

\[ \text{boole-nond} \rightarrow (\text{int-a} \ \& \ \neg \text{int-b}) \]

\[ \text{boole-ior} \rightarrow \text{int-a} \lor \text{int-b} \]

\[ \text{boole-andc1} \rightarrow \text{int-a} \land \text{int-b} \]

\[ \text{boole-andc2} \rightarrow \text{int-a} \land \text{int-b} \]

\[ \text{boole-nott} \rightarrow (\text{int-a} \ \& \ \neg \text{int-b}) \]

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

\[ (\text{logeqv} \text{ integer}^{*}) \]

\[ (\text{logand} \text{ integer}^{*}) \]

\[ \rightarrow \text{Return value of exclusive-or of anded integers, respectively. Without any integer, return 1.} \]

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

\[ \rightarrow \text{int-a} \land \text{int-b} \]

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

\[ \rightarrow \text{int-a} \land \text{int-b} \]

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

\[ \rightarrow (\text{int-a} \ \& \ \neg \text{int-b}) \]

\[ (\text{logxor} \text{ integer}^{*}) \]

\[ \rightarrow \text{Return value of exclusive-or of anded integers, respectively. Without any integer, return 0.} \]

\[ (\text{logorc1} \text{ int-a} \text{ int-b}) \]

\[ \rightarrow \text{int-a} \lor \text{int-b} \]

\[ (\text{logorc2} \text{ int-a} \text{ int-b}) \]

\[ \rightarrow \text{int-a} \lor \text{int-b} \]

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

\[ \rightarrow (\text{int-a} \ \& \ \neg \text{int-b}) \]

\[ (\text{logbtp} \text{ i int}) \]

\[ \rightarrow \text{T if zero-indexed nth bit of int is set.} \]

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

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

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

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

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

- Number of bits necessary to represent integer.

\(\text{\texttt{(ldb-test byte-spec integer)}}\)

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

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

- Return copy of \texttt{integer} arithmetically shifted left or right by \texttt{count}

\(\text{\texttt{(ldb byte-spec integer)}}\)

- \texttt{Extract byte denoted by \texttt{byte-spec} from integer. set-table.}

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

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

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

- Return copy of \texttt{integer} with all bits unset but those denoted by \texttt{byte-spec. set-table.}

\(\text{\texttt{(byte size position)}}\)

- \texttt{Byte specifier for a byte of size \texttt{bits} starting at a weight of \texttt{position}.}

\(\text{\texttt{(byte-size byte-spec)}}\)

- \texttt{Size or position, respectively, of byte-spec.}

1.5 Implementation-Dependent

\text{\texttt{short-float}}

\text{\texttt{single-float}}

\text{\texttt{double-float}}

\text{\texttt{long-float}}

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

\text{\texttt{least-negative}}

\text{\texttt{least-negative-normalized}}

\text{\texttt{least-positive-normalized}}

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

\text{\texttt{most-negative}}

\text{\texttt{most-positive}}

\text{\texttt{fixnum}}

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

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

- \texttt{(integer-decode-float n)}

\(\text{\texttt{(scale-float \alpha)}}\)

- With \(n\)’s radix \(b\), return \(nb^\alpha\).

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

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

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

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

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

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

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

\((\text{string} \ foo)\)
\(\triangleright\) T if foo is of indicated type.

\((\text{simple-string-p} \ foo)\)
\(\triangleright\) T if foo is a capitalized string, a subset of words, an all-capitalized version of implementation, operating system, or hardware, respectively.

\((\text{string= string-equal})\)
\(\triangleright\) Return T if subsequences of foo and bar are equal. Obey/ignore, respectively, case.

\((\text{string|=} \ not-equal)\)
\(\triangleright\) 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.

\((\text{string>= greaterp})\)
\(\triangleright\) Return start-string bar end-string if subsequences of foo and bar are greater.

\((\text{string<= lessp})\)
\(\triangleright\) Return start-string bar end-string if subsequences of foo and bar are less.

\((\text{string<= not-greaterp})\)
\(\triangleright\) Return start-string bar end-string if subsequences of foo and bar are not greater.

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

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

\((\text{string-capitalize})\)
\(\triangleright\) Convert string into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

\((\text{string-upcase})\)
\(\triangleright\) Convert string to uppercase.

\((\text{string-downcase})\)
\(\triangleright\) Convert string to lowercase.

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

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

\((\text{parse-integer string})\)
\(\triangleright\) Return integer parsed from string and index of parse end.

4 Conses

4.1 Predicates

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

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

\((\text{endp list})\)
\(\triangleright\) Return T if list/fo is NIL.

\((\text{null foo})\)
\(\triangleright\) Return T if foo is NIL.

16 External Environment

\((\text{get-internal-real-time})\)
\(\triangleright\) Current time, or computing time, respectively, in clock ticks.

\((\text{get-internal-run-time})\)
\(\triangleright\) Number of clock ticks per second.

\((\text{get-internal-time-units-per-second})\)
\(\triangleright\) Seconds from 1900-01-01, 00:00, ignoring leap seconds.

\((\text{get-universal-time})\)
\(\triangleright\) Seconds from 1900-01-01, 00:00, ignoring leap seconds.

\((\text{get-decoded-time})\)
\(\triangleright\) Return second, minute, hour, date, month, year, day, daylight-p, and zone.

\((\text{short-site-name})\)
\(\triangleright\) String representing physical location of computer.

\((\text{long-site-name})\)
\(\triangleright\) Name or version of implementation, operating system, or hardware, respectively.

\((\text{machine-instance})\)
\(\triangleright\) Computer name.
15.4 Declarations

\texttt{(\texttt{proclaim decl})}
\begin{itemize}
\item \texttt{declare decl*}}
\end{itemize}
\begin{itemize}
\item \texttt{declare decl*}}
\end{itemize}
\begin{itemize}
\item \texttt{declare decl*}}
\end{itemize}
\begin{itemize}
\item \texttt{declaration foo*}}
\end{itemize}
\begin{itemize}
\item \texttt{(dynamic-extent variable* \texttt{(function function*)}}
\end{itemize}
\begin{itemize}
\item \texttt{(type type variable*)}
\end{itemize}
\begin{itemize}
\item \texttt{(ftype type function*)}
\end{itemize}
\begin{itemize}
\item \texttt{declare variables or functions to be of type.}
\end{itemize}
\begin{itemize}
\item \texttt{declare variables or functions to be of type.}
\end{itemize}
\begin{itemize}
\item \texttt{suppress warnings about used/unused bindings.}
\end{itemize}
\begin{itemize}
\item \texttt{tell compiler to integrate/not to integrate, respectively, called functions into the calling routine.}
\end{itemize}
\begin{itemize}
\item \texttt{make foos names of declarations.}
\end{itemize}
\begin{itemize}
\item \texttt{declare lifetime of variables and/or functions to end when control leaves enclosing block.}
\end{itemize}
\begin{itemize}
\item \texttt{if foo is a tail of list, return preceding part of list. Otherwise return list.}
\end{itemize}
\begin{itemize}
\item \texttt{if foo is already member of list. If not, return list.}
\end{itemize}
\begin{itemize}
\item \texttt{set place to (cdr place), return (car place).}
\end{itemize}
4.3 Association Lists

(maplis 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 first cons whose car, or cdr, respectively, satisfies test.

assoc-if-not)  ▷ Return list of values of form and an list or, with only one argument, foo, foo can be of any type.

reappend list foo)  ▷ Return concatenated list after reversing order in list.

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

4.4 Trees

(tree-equal foo bar)  ▷ 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.

macroexpand-1 form)  ▷ Return macro expansion, once or entirely, respectively, of form and T if form was a macro form. Return form and NIL otherwise.
15 Compiler

15.1 Predicates

\( \texttt{special-operator-p foo} \) \( \iff \) foo is a special operator.

\( \texttt{compiled-function-p foo} \) \( \iff \) foo is of type compiled-function.

15.2 Compilation

\( \texttt{compile \{ \begin{array}{l} \texttt{NIL \ definition} \\ \{ \texttt{name} \ (\texttt{set \ name}) \ \texttt{[definition]} \end{array} \} } \)

\( \iff \) Return compiled function or replace name’s function definition with the compiled function. Return \( \top \) in case of warnings or errors, and \( \bot \) in case of warnings or errors excluding style-warnings.

\( \texttt{compile-file \ file \ \{ \begin{array}{l} \texttt{:output-file \ out-path} \\ \texttt{:verbose \ book} \\ \texttt{:print \ book} \\ \texttt{:+compile-print} \\ \texttt{:external-format \ file-format \ default} \end{array} \} } \)

\( \iff \) Write compiled contents of file to out-path. Return true output path or NIL, \( \top \) in case of warnings or errors, \( \bot \) in case of warnings or errors excluding style-warnings.

\( \texttt{compile-file-pathname \ file \ :output-file \ path \ [\texttt{other-keyargs}] } \)

\( \iff \) Pathname \texttt{compile-file} writes to if invoked with the same arguments.

\( \texttt{load \ path \ \{ \begin{array}{l} \texttt{:verbose \ book} \\ \texttt{:print \ book} \\ \texttt{:load-print} \\ \texttt{:if-does-not-exist \ bool} \\ \texttt{:external-format \ file-format \ default} \end{array} \} } \)

\( \iff \) Load source file or compiled file into Lisp environment. Return \( \top \) if successful.

14.4 Standard Packages

\texttt{common-lisp}

\( \iff \) Exports the defined names of Common Lisp except for those in the \texttt{keyword} package.

\texttt{common-lisp-user|d-user}

\( \iff \) Current package after startup; uses package \texttt{common-lisp}.

\texttt{keyword}

\( \iff \) Contains symbols which are defined to be of type \texttt{keyword}.

5 Arrays

5.1 Predicates

\( \texttt{arrayp foo} \)

\( \iff \) \texttt{foo} is of indicated type.

\( \texttt{simple-vector-p foo} \)

\( \iff \) \texttt{foo} is a simple vector.

\( \texttt{bit-vector-p foo} \)

\( \iff \) \texttt{foo} is a bit vector.

\( \texttt{adjustable-array-p array} \)

\( \iff \) \texttt{array} is adjustable/has a fill pointer, respectively.

\( \texttt{array-in-bounds-p array [subscripts]} \)

\( \iff \) Return \( \top \) if \texttt{subscripts} are in \texttt{array}’s bounds.

5.2 Array Functions

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

\( \iff \) Return an \texttt{array} of indicated type.

\( \texttt{adjust-array \ dimension-sizes \ [adjustable \ bool]} \)

\( \iff \) Return \texttt{array} of indicated type.

\( \texttt{element-type array} \)

\( \iff \) Initial element of \texttt{array}.

\( \texttt{fill-pointer \ array \ [num \ bool]} \)

\( \iff \) Initial contents of \texttt{array}.

\( \texttt{displaced-to \ array \ [displaced-index-offset]} \)

\( \iff \) Return fresh, or readjust, respectively, \texttt{vector} or \texttt{array}.

\( \texttt{arref \ array \ [subscripts]} \)

\( \iff \) Return \texttt{array} pointed to by \texttt{subscripts}.

\( \texttt{row-major-arref \ array \ i} \)

\( \iff \) Return \texttt{ith} element of \texttt{array} in row-major order.

\( \texttt{array-dimensions \ array} \)

\( \iff \) List containing the lengths of \texttt{array}’s dimensions.

\( \texttt{array-dimension \ array \ i} \)

\( \iff \) Length of \texttt{ith} dimension of \texttt{array}.

\( \texttt{array-total-size \ array} \)

\( \iff \) Number of elements in \texttt{array}.

\( \texttt{array-rank \ array} \)

\( \iff \) Number of dimensions of \texttt{array}.

\( \texttt{array-displacement \ array} \)

\( \iff \) Target array and offset.
\( (\, \text{bit} \, \text{array} \, [\text{subscripts}] \) \\
\( (\, \text{bit} \, \text{simple-bit-array} \, [\text{subscripts}] \) \\
\( \triangleright \) Return \textit{element} of \textit{bit-array} or of \textit{simple-bit-array}. \textit{setf}able.

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

\( (\, \text{bit-eqv} \, \text{bit-and} \, \text{bit-andc1} \, \text{bit-andc2} \, \text{bit-nand} \, \text{bit-or} \, \text{bit-orc1} \, \text{bit-orc2} \, \text{bit-xor} \, \text{bit-nor} \) \)

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

\( \text{array-rank-limit} \) \( \triangleright \) Upper bound of array rank; \( \geq 8 \).

\( \text{array-dimension-limit} \) \( \triangleright \) Upper bound of an array dimension; \( \geq 1024 \).

\( \text{array-total-size-limit} \) \( \triangleright \) Upper bound of array size; \( \geq 1024 \).

5.3 \textbf{Vector Functions}

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

\( (\, \text{vector} \, \text{foo} \) \\
\( \triangleright \) Return fresh simple vector of \textit{foos}.

\( (\, \text{sref} \, \text{vector} \, \text{i} \) \\
\( \triangleright \) Element \( i \) of simple vector. \textit{setf}able.

\( (\, \text{vector-push} \, \text{foo} \, \text{vector} \) \\
\( \triangleright \) Return \text{NIL} if \textit{vector}'s fill pointer equals size of \textit{vector}. Otherwise replace element of \textit{vector} pointed to by \textit{fill pointer} with \textit{foo}; then increment fill pointer.

\( (\, \text{vector-push-extend} \, \text{foo} \, \text{vector} \, [\text{num}] \) \\
\( \triangleright \) Replace element of \textit{vector} pointed to by \textit{fill pointer} with \textit{foo}, then increment \textit{fill pointer}. Extend \textit{vector}'s size by \( \geq \text{num} \) if necessary.

\( (\, \text{vector-pop} \, \text{vector} \) \\
\( \triangleright \) Return element of \textit{vector} its fillpointer points to after decrementation.

\( (\, \text{fill-pointer} \, \text{vector} \) \\
\( \triangleright \) Fill pointer of \textit{vector}. \textit{setf}able.

6 \textbf{Sequences}

6.1 \textbf{Sequence Predicates}

\( (\, \text{every} \, \text{notevery} \) test \textit{sequence} \)

\( \triangleright \) Return \text{NIL} or \( T \), respectively, as soon as \textit{test} on any set of corresponding elements of \textit{sequences} returns \text{NIL}.

\( (\, \text{some} \, \text{notany} \) test \textit{sequence} \)

\( \triangleright \) Return value of \textit{test} or \text{NIL}, respectively, as soon as \textit{test} on any set of corresponding elements of \textit{sequences} returns non-\text{NIL}.

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

\( (\, \text{export} \, \text{symbols} \, \text{package} \, \text{package} \) \\
\( \triangleright \) Make symbols external to \textit{package}. Return \( T \).

\( (\, \text{unexport} \, \text{symbols} \, \text{package} \, \text{package} \) \\
\( \triangleright \) Revert symbols to internal status. Return \( T \).

\( (\, \text{do-symbols} \, \text{do-external-symbols} \, \text{do-all-symbols} \) \)

\( \{(\, \text{declare} \, \text{decl} \, \text{form} \) \}

\( \triangleright \) Evaluate \textit{tagbody}-like body with \text{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 \text{block} named \text{NIL}.

\( (\, \text{with-package-iterator} \, \text{foo} \, \text{packages} \, \text{[internal external inherited]} \)

\( (\, \text{declare} \, \text{decl} \, \text{form} \) \)

\( \triangleright \) Return values of \textit{forms}. In \textit{forms}, successive invocations of \text{foo} return \( T \) if a symbol is returned; a symbol from \textit{packages}; accessibility (\text{internal}, \text{external}, or \text{inherited}); and the package the symbol belongs to.

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

\( (\, \text{provide} \, \text{module} \) \\
\( \triangleright \) If not already there, add \text{module} to \text{*modules*}. Deprecated.

\( \text{*modules*} \) \( \triangleright \) List of names of loaded modules.

14.3 \textbf{Symbols}

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

\( (\, \text{make-symbol} \, \text{name} \) \\
\( \triangleright \) Make fresh, uninterned \text{symbol} \text{name}.

\( (\, \text{gensym} \, [\text{#*}] \) \\
\( \triangleright \) Return fresh, uninterned symbol \#*\text{sn} with \text{sn} from \text{gensym-counter*}. Increment \text{*gensym-counter*}.

\( (\, \text{gentemp} \, [\text{prefix} \, \text{package} \, \text{package}] \) \\
\( \triangleright \) Intern fresh symbol in \text{package}. Deprecated.

\( (\, \text{copy-symbol} \, \text{symbol} \, [\text{property}] \) \\
\( \triangleright \) Return uninterned copy of \textit{symbol}. If \text{props} is \( T \), give copy the same value, function and property list.

\( (\, \text{symbol-name} \, \text{symbol} \) \\
\( \triangleright \) Return \text{symbol}'s \text{name}.

\( (\, \text{symbol-package} \, \text{symbol} \) \\
\( \triangleright \) Return \text{symbol}'s \text{package}.

\( (\, \text{symbol-plist} \, \text{symbol} \) \\
\( \triangleright \) Return \text{symbol}'s \text{plist}.

\( (\, \text{symbol-value} \, \text{symbol} \) \\
\( \triangleright \) Get/set \text{symbol}'s \text{value}.

\( (\, \text{documentation} \, \text{setf} \, \text{documentation} \, \text{new-doc} \, \text{foo} \) \)

\( \{\, \text{variable} \, \text{function} \, \text{compiler-macro} \, \text{method-combination} \, \text{structure} \, \text{type} \, \text{setf} \) \)

\( \triangleright \) Get/set \text{documentation} string of \textit{foo} of given type.
14.2 Packages

▷ bar
  ▷ Keyword, evaluates to :bar.

package:symbol
▷ Exported symbol of package.

package::symbol
▷ Possibly unexported symbol of package.

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

(make-package foo {displaynicknames (nicks)}
  {displayuse (used-package)})
▷ 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 package})
▷ Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.

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

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

*package*{common-lisp-user}
▷ 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 symbol package)
{displaypackage}
▷ 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).

(find-symbol package)
{displaypackage}
▷ Remove symbol from package, return T on success.

(import symbols package)
{displaypackage}
▷ 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)
{displaypackage}
▷ 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 foo sequence {start start} {end end})
▷ Return number of elements in sequence which match foo.

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

(count-if-not test sequence {start start} {end end})
▷ Return number of elements in sequence which do not satisfy test.

(subseq sequence start {end end})
▷ Return subsequence of sequence between start and end.

(reverse sequence)
▷ Return sequence in reverse order.
(#'eql pathname string that sufficiently describes the type.

(path-or-stream path string that is a legal path name.

(exists path-or-stream) Return true if path is a legal path name.

(logical-pathname logical-path-or-stream) Logical path name.

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

(load-logical-pathname-translations logical-host) Load logical-host's translations.

(translate-logical-pathname path-or-stream) Physical path name.

(probe-file file) Canonical name of file.

(truename file) Canonical name.

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

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

(file-length stream) Length of stream.

(rename-file foo bar) Rename file foo to bar.

(ensure-directories-exist path [verbose bool]) Create parts of path if necessary.

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

(map-into result-sequence function sequence*)
▷ Store into result-sequence successively 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
    :test (eq | eql | equal | equalp) | n eq)
▷ 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. setfable.

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

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

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

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

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

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

(hash-table-size hash-table)

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

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

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

(defstruct (foo bar)
  (conc-name [nil])
  (constructor [nil])
  (copier [nil])
)

(include struct (slot-init :type (read-only bo))
  (type list vector (vector type))
  (print-object [o-printer])
  (print-function [f-printer])
  (predicate [p-name [nil]])

> Define structure foo together with functions MAKE-foo, COPY-foo and foo-P; and setfable accessors 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-λ (see page 18) is given, by (maker arg" {key value}'}. In the latter case, args and :keys correspond to the positional and keyword parameters defined in ord-λ whose vars in turn correspond to slots. :print-object/:print-function generate a print-object method for an instance bar of foo calling (o-printer bar stream) 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 equal; or are the same character ignoring case; or are numbers of the same value ignoring type; or are equivalent pathnames; or are conses or arrays of the same shape with eql elements; or are structures of the same type with eql elements; or are hash-tables of the same size with the same :test function, the same keys in terms of :test function, and eql elements.

(not foo)  > T if foo is NIL; NIL otherwise.

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

(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 synonym-stream)
> Return a string-stream accepting characters available via get-output-stream-string.

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

(two-way-stream-input-stream two-way-stream)
(two-way-stream-output-stream two-way-stream)
(choo-echo-stream-input-stream echo-stream)
(choo-echo-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-stream)
> Clear and return as a string characters on string-stream.

(file-position-stream (start 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 string.

(listen stream standard-input)
> T if there is a character in input stream.

(clar-output 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) (declare decl*)* form*)
> Evaluate forms with foo locally bound to input string-stream from string. Return values of forms; store next reading position into index.
Go-To. Jump m arguments forward, or backward, or to argument n.

- [n; |]- [n; |:] T
  ▪ Indent. Set indentation to n relative to leftmost/to current position.

- (\[ i; |]- [i; |:] t) T
  ▪ Tabulate. Move cursor forward to column number c + ki, k ≥ 0 being as small as possible. With ;, calculate column numbers relative to the immediately enclosing section. With \( \Theta \), move to column number \( c_0 + c' + ki \) where \( c_0 \) is the current position.

- [n; |]\[ - [n; |]: - [n; |: \[ t \] \[ \] T
  ▪ Go-To. Jump m arguments forward, or backward, or to argument n.

- [t] \[ i; |]\[ \{ text - \] T
  ▪ Iteration. Use text repeatedly, up to limit, as control string for the elements of the list argument or (with \( \Theta \)) for the remaining arguments. With ; or \( \Theta \), list elements or remaining arguments should be lists of which a new one is used at each iteration step.

- [x [y [z]]]
  ▪ Escape Upward. Leave immediately \( < - > \), \( < - : > \), \( - \{- \}, - ? \), or the entire \( \gamma \)format operation. With one to three prefixes, act only if \( x = 0 \), \( x = y \), or \( x ≤ y ≤ z \), respectively.

- [i [\[ ]\[ \{ text - \}] text [-; default -] T
  ▪ Conditional Expression. Use the zero-indexed argument (or ith given) text as a \( \gamma \)format control subclause. With ;, use the first text if the argument value is Nil, or the second text if it is T. With \( \Theta \), do nothing for an argument value of Nil. Use the only text and leave the argument to be read again if it is T.

- [-\[ ; !? \] T
  ▪ Recursive Processing. Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

- [i [\[ ]\[ \{ package - \}[\[ \{ \} \] \[ \} \] \[ function - \} \[ \] T
  ▪ Call Function. Call all-uppercsse package:function with the arguments stream, format-argument, colon-p, at-sign-p and prefixes for printing format-argument.

- [i [\[ ]\[ \{ text - \}] text [-; default -] T
  ▪ Write. Print argument of any type obeying every printer control variable. With ;, pretty-print. With \( \Theta \), print without limits on length or depth.

- [\[ \] ; #]
  ▪ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams

\[
\begin{align*}
&\{\text{open path}\} \\
&\text{:direction} \quad \text{input} \quad \text{output} \quad \text{read} \quad \text{write} \\
&\text{:element-type} \quad \text{type} \quad \text{default} \quad \text{symbol} \quad \text{package} \quad \text{function} \\
&\text{:new-version} \quad \text{error} \\
&\text{:if-exists} \quad \text{name} \quad \text{rename} \quad \text{format} \\
&\text{:if-does-not-exist} \quad \text{create} \quad \text{error} \quad \text{NIL} \\
&\text{:external-format} \quad \text{file-stream} \quad \text{form} \\
\end{align*}
\]

\( (/\text{constantp} \ foo \ [\text{environment} \ \text{nil}] ) \)
  ▪ T if foo is a constant form.

\( (/\text{functionp} \ foo ) \)
  ▪ T if foo is of type function.

\( (/\text{boundp} \ foo \ (\text{setf} \ foo ) ) \)
  ▪ T if foo is a global function or macro.

9.2 Variables

\( (/\text{defconstant} \ foo \ form \ (\text{doc}) ) \)
  ▪ Assign value of form to global constant/dynamic variable foo.

\( (/\text{defparameter} \ foo \ form \ (\text{doc}) ) \)
  ▪ Unless bound already, assign value of form to dynamic variable foo.

\( (/\text{set} \ \text{symbol} \ foo ) \)
  ▪ Set symbol’s value cell to foo. Deprecated.

\( (/\text{multiple-value-set} \ vars \ form ) \)
  ▪ Set elements of vars to the values of form. Return form’s primary value.

\( (/\text{shift} \ \text{place} + \ foo ) \)
  ▪ Store value of foo in rightmost place shifting values of places left, returning first place.

\( (/\text{rotatef} \ \text{place} ^ * ) \)
  ▪ Rotate values of places left, old first becoming new last place’s value. Return NIL.

\( (/\text{makunbound} \ \text{foo} ) \)
  ▪ Delete special variable foo if any.

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

\( (/\text{get-properties} \ \text{property-list} \ \text{keys} ) \)
  ▪ 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, \( \text{nil} \), and \( \text{nil} \) if there was no matching key in property-list.

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

\( (/\text{progv} \ \text{symbols} \ \text{values} \ \text{form} ^ * ) \)
  ▪ Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.

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

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

\[
\left(\begin{array}{l}
\text{var}^* \\
\text{optional}
\end{array}\right) \text{ var } \left(\begin{array}{c}
\text{form}^* \\
\text{body-forms}^*
\end{array}\right)
\]

- \text{defun}\ (\text{name}) \text{ (form)}\ \left(\begin{array}{l}
\text{declare}\ \text{decl}^* \\
\text{form}^* \\
\text{body}^*
\end{array}\right)
- \text{lambda}\ (\text{ord-λ})\ \left(\begin{array}{c}
\text{form}^* \\
\text{body}^*
\end{array}\right)

\begin{itemize}
\item Define a function named \text{name} or (setf \text{name}), or an anonymous function, respectively, which applies \text{form} to \text{ord-λ}. For \text{defun}, forms are enclosed in an implicit \text{block} named \text{name}.
\item \text{lambda} (\text{ord-λ}) \text{ form}^* \text{ body}^* defines a function named \text{name} or (setf \text{name}), or an anonymous function, respectively, which applies \text{form} to \text{ord-λ}. For \text{defun}, forms are enclosed in an implicit \text{block} named \text{name}.
\end{itemize}

\[
\left\{\begin{array}{l}
\text{fun}^* \\
\text{function}
\end{array}\right\}
\]

\[\begin{array}{l}
\text{return}\ \text{function}\ \text{arg}^* \text{ args}
\end{array}\]

\[
\left(\begin{array}{c}
\text{values-list}\ \text{list} \\
\text{values}\ \text{foo}
\end{array}\right)
\]

\[
\begin{array}{l}
\text{multiple-value-list}\ \text{form}^* \\
\text{nth-value}\ n\ \text{form}
\end{array}\]

\[
\begin{array}{l}
\text{complement}\ \text{function}
\end{array}\]

- \text{return}\ \text{function}\ \text{arg}^* \text{ args}
- \text{return}\ \text{function}\ \text{arg}^* \text{ args}
- \text{return}\ \text{values-list}\ \text{list} \\
- \text{return}\ \text{values}\ \text{foo}
- \text{return}\ \text{multiple-value-list}\ \text{form}^* \\
- \text{return}\ \text{nth-value}\ n\ \text{form}
- \text{return}\ \text{complement}\ \text{function}

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

\[
\begin{array}{l}
\text{function}\ \left(\begin{array}{c}
\text{values-list}\ \text{list} \\
\text{values}\ \text{foo}
\end{array}\right)
\end{array}\]

\[
\begin{array}{l}
\text{multiple-value-call}\ \text{function}\ \text{form}^* \\
\text{call}\ \text{function}\ \text{with}\ \text{all}\ \text{the}\ \text{values}\ \text{of}\ \text{each}\ \text{form}\ \text{as}\ \text{its}\ \text{arguments}.
\end{array}\]

\[
\begin{array}{l}
\text{return}\ \text{elements}\ \text{of}\ \text{list} \\
\text{return}\ \text{multiple-values}\ \text{values}\ \text{of}\ \text{the}\ \text{primary}\ \text{values}\ \text{of}\ \text{the}\ \text{foos}.\ \text{setfable}
\end{array}\]

\[
\begin{array}{l}
\text{return}\ \text{zero-indexed}\ \text{nth}\ \text{return}\ \text{value}\ \text{of}\ \text{form}.
\end{array}\]

\[
\begin{array}{l}
\text{return}\ \text{new}\ \text{function}\ \text{with}\ \text{same}\ \text{arguments}\ \text{and}\ \text{same}\ \text{side}\ \text{effects}\ \text{as}\ \text{function},\ \text{but}\ \text{with}\ \text{complementary}\ \text{truth}\ \text{value}.
\end{array}\]

\[
\begin{array}{l}
\text{fixed-format-floating-point}.\ \text{with}\ \text{T},\ \text{always}\ \text{prepend}\ \text{a}\ \text{sign}.
\end{array}\]

\[
\begin{array}{l}
\text{exponential-general-floating-point}.\ \text{print}\ \text{argument}\ \text{as}\ \text{floating-point}\ \text{number}\ \text{with}\ \text{dec-digits}\ \text{after}\ \text{decimal}\ \text{point}\ \text{and}\ \text{exp-digits}\ \text{in}\ \text{the}\ \text{signed}\ \text{exponent}.\ \text{with}\ -\text{G},\ \text{choose}\ \text{either}\ \text{-E}\ \text{or}\ \text{-F}.\ \text{with}\ \text{T},\ \text{always}\ \text{prepend}\ \text{a}\ \text{sign}.
\end{array}\]

\[
\begin{array}{l}
\text{case-conversion}.\ \text{convert}\ \text{text}\ \text{to}\ \text{lowercase},\ \text{convert}\ \text{first}\ \text{letter}\ \text{of}\ \text{each}\ \text{word}\ \text{to}\ \text{uppercase},\ \text{capitalize}\ \text{first}\ \text{word}\ \text{and}\ \text{convert}\ \text{the}\ \text{rest}\ \text{to}\ \text{lowercase},\ \text{or}\ \text{convert}\ \text{to}\ \text{uppercase},\ \text{respectively}.
\end{array}\]

\[
\begin{array}{l}
\text{plural}.\ \text{if}\ \text{argument}\ \text{eq}\ \text{1}\ \text{print}\ \text{nothing},\ \text{otherwise}\ \text{print}\ \text{a},\ \text{do}\ \text{the}\ \text{same}\ \text{for}\ \text{the}\ \text{previous}\ \text{argument};\ \text{if}\ \text{argument}\ \text{eq}\ \text{1}\ \text{print}\ \text{y},\ \text{otherwise}\ \text{print}\ \text{ies};\ \text{do}\ \text{the}\ \text{same}\ \text{for}\ \text{the}\ \text{previous}\ \text{argument},\ \text{respectively}.
\end{array}\]

\[
\begin{array}{l}
\text{conditional-newline}.\ \text{print}\ \text{newline}\ \text{like}\ \text{pprint-newline}\ \text{with}\ \text{argument}\ \text{dlinear},\ \text{fill},\ \text{miser},\ \text{or}\ \text{mandatory},\ \text{respectively}.
\end{array}\]

\[
\begin{array}{l}
\text{page}.\ \text{print}\ \text{n}\ \text{page}\ \text{separators}.
\end{array}\]

\[
\begin{array}{l}
\text{tildes}.\ \text{print}\ \text{n}\ \text{tildes}.
\end{array}\]

\[
\begin{array}{l}
\text{justification}.\ \text{justify}\ \text{text}\ \text{produced}\ \text{by}\ \text{texts}\ \text{in}\ \text{field}\ \text{of}\ \text{at}\ \text{least}\ \text{max-col}\ \text{columns}.\ \text{with}\ \text{T},\ \text{right}\ \text{justify};\ \text{with}\ \text{NIL},\ \text{left}\ \text{justify}.\ \text{if}\ \text{this}\ \text{would}\ \text{leave}\ \text{less}\ \text{than}\ \text{max-col}\ \text{characters}\ \text{on}\ \text{the}\ \text{current}\ \text{line},\ \text{output}\ \text{nl-text}\ \text{first}.
\end{array}\]

\[
\begin{array}{l}
\text{logical-block}.\ \text{act}\ \text{like}\ \text{pprint-logical-block}\ \text{using}\ \text{body}\ \text{as}\ \text{format}\ \text{control}\ \text{string}\ \text{on}\ \text{the}\ \text{elements}\ \text{of}\ \text{the}\ \text{list}\ \text{argument},\ \text{or},\ \text{with}\ \text{T},\ \text{on}\ \text{the}\ \text{remaining}\ \text{arguments},\ \text{which}\ \text{are}\ \text{extracted}\ \text{by}\ \text{pprint-pop}.\ \text{with}\ \text{T},\ \text{prefix}\ \text{and}\ \text{suffix}\ \text{default}\ \text{to}\ \text{and}\ \text{and}\ \text{).}\ \text{when}\ \text{closed}\ \text{by}\ \text{T}\text{ body}\ \text{are}\ \text{replaced}\ \text{with}\ \text{conditional}\ \text{newlines}.
\end{array}\]
+print-case+ (upcase) ➔ 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 prettily.

+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

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

(format [nil] [out-string] [out-string] control arg*) ➔ Output control string which may contain - directives possibly taking args. Alternatively, control can be a function returned by setformat which is then applied to out-string and arg*. Output to out-string, out-stream or, if first argument is T, to *standard-output*. Return NIL. If first argument is NIL, return formatted output.

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

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

(*constantly foo) ➔ Function of any number of arguments returning foo.

(*identity foo) ➔ Return foo.

(*function-lambda-expression function) ➔ If available, return lambda expression of function, NIL if function was defined in an environment without bindings, and name of function.

(*definition (foo (setffoo))) ➔ Definition of global function foo. settable.

(*makunbound foo) ➔ Remove global function or macro definition foo.

+call-arguments-limit+ ➔ Upper bound of the number of function arguments or lambda list parameters, respectively; ≥ 50.

+multiple-values-limit+ ➔ Upper bound of the number of values a multiple value can have; ≥ 20.

9.4 Macros

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

[&whole var] [E] [var (macro-λ*)] [E]

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

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

[&allow-other-keys] [&aux [var init-supplied-p]] [E] or

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

[&aux [var (macro-λ*)] [init-supplied-p]] [E] , rest-var].

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.

(defmacro (defmacro-compiler-macro) (foo (setf foo) (macro-λ*) (declare decl*) [doc] form*) ➔ Define macro foo which on evaluation as (foo tree) applies expanded forms to arguments from tree, which corresponds to tree-shaped macro-λs. forms are enclosed in an implicit block named foo.

define-symbol-macro foo form) ➔ Define symbol macro foo which on evaluation evaluates expanded form.

(macrolet ((foo (macro-λ*) (declare decl*) [doc] form*) ➔ Evaluate forms with locally defined mutually invisible macros foo which are enclosed in implicit blocks of the same name.
(symbol-macrolet ((foo expansion-form*) (declare decl*) form*))
  ▷ Evaluate forms with locally defined symbol macros foo.

(defun defsetf function
  updater [doc]
  ([self-λ] (s-var*) (declare decl*) [doc] form*)
) where defsetf lambda list (setf-λ*) has the form
  (var* &optional [var [init nil] [supplied-p] (])]
  (&rest var) &key (var [init nil] [supplied-p] (])]
  (&allow-other-keys &environment var))
▷ Specify how to set a place accessed by function.
Short form: (setf (function 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-λ 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-λ*) (declare decl*) [doc] form*)
▷ Specify how to set a place accessed by function. On invocation of (setf (function arg*) value-form); form* must expand into code returning args, args, neutral-vars, set-form, and get-form as described with get-set-expansion where the elements of macro lambda list macro-λ* are bound to corresponding args. forms are enclosed in an implicit block named function.

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

(define-modify-macro foo (&optional
  [var [init nil] [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 &key 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.

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

(defun print-fill stream foo [parenthesis nil] [noop])
(defun print-tabular stream foo [parenthesis nil] [noop nil])
(defun print-linear stream foo [parenthesis nil])
▷ 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. Use with *format directive* /-

(defun print-logical-block stream list
  ▷ 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.

(defun print-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.

(defun print-newline stream
  ▷ Print a conditional newline if stream is a pretty printing stream. Return NIL otherwise.

;array bool ;base radix
;caps case
;downcase upcase
;circle bool ;escape bool
;capitalize ;gensym bool
;length int nil
;level int nil
;lines int nil
;miser-width int nil
;pprint-dispatch dispatch-table
;pretty bool ;radix bool
;readably bool ;right-margin int nil
;stream stream ;standard-output

;&array bool
;base radix
;case downcase
;circle boolean
;escape boolean
;capitalize ;gensym boolean
;length int nil
;level int nil
;lines int nil
;miser-width int nil
;pprint-dispatch dispatch-table
;pretty boolean
;radix boolean
;readably boolean
;right-margin int nil
;stream stream ;standard-output

;*print-array* ▷ If T, print arrays readably.
;*print-base* 16 ▷ Radix for printing rationals, from 2 to 36.
#+feature when-feature

#-feature unless-feature

▷ Means when-feature if feature is T; means unless-feature if feature is NIL. Feature is a symbol from `*features*, or `*{and | or} feature*`, or `*{not feature}*`.

\*features*

▷ List of symbols denoting implementation-dependent features.

\[c^*]\):

▷ Treat arbitrary character(s) \(c\) as alphabetic preserving case.

13.4 Printer

\[
\begin{align*}
\text{\textbf{(prin1 \{ prin \ pprint \ prin1-to-string \ prin1-to-string \})}} \\
\text{\textbf{foo \|stream \standard-output\}}} \\
\text{▷ Print foo to stream \textit{readably}, \textit{readably} between a newline and a space, \textit{readably} after a newline, or human-readable without any extra characters, respectively. prin1, prin and } \\
\text{prin1-to-string return foo.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(print-object object \stream)}} \\
\text{▷ Print object to stream. Called by the Lisp printer.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(print-unreadable-object \{ foo \|stream \|:type \boo | \:identity \boo | \| form \}}} \\
\text{▷ Enclosed in \#< and \#>, print foo by means of forms to stream. Return NIL.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(terpri \|stream \|standard-output\)}} \\
\text{▷ Output a newline to stream. Return NIL.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(fresh-line \|stream \|standard-output\)}} \\
\text{▷ Output a newline to stream and return T unless stream is already at the start of a line.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(write-char char \|stream \|standard-output\)}} \\
\text{▷ Output \textit{char} to \textit{stream}.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(write-string string \|stream \|standard-output\)}} \\
\text{▷ Write \textit{string} to \textit{stream} without/with a trailing newline.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(write-byte byte \|stream \)}} \\
\text{▷ Write \textit{byte} to binary \textit{stream}.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(write-sequence sequence \|stream \)}} \\
\text{▷ Write elements of \textit{sequence} to binary or character \textit{stream}.} \\
\end{align*}
\]

9.5 Control Flow

\[
\begin{align*}
\text{\textbf{(if test then \{ else \}}} \\
\text{▷ Return values of} \textit{then}\text{if} test\text{returns T; return values of} \textit{else} otherwise.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(when test)}} \\
\text{▷ Evaluate} foo\text{and return their values if} test\text{returns T or NIL, respectively. Return NIL otherwise.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(case test \{}} \\
\text{▷ Return the values of the first} f o o^*\text{one of whose keys is} eql\text{test. Return values of} f o o^*\text{if there is no matching key.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(and form \)}} \\
\text{▷ Evaluate forms from left to right. Immediately return NIL} \text{if one form\text{’}s value is NIL. Return values of last form\text{, otherwise.}} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(or form \)}} \\
\text{▷ Evaluate forms from left to right. Immediately return primary value of first non-NIL-evaluating form, or all values} \text{if last form is reached. Return NIL if no form returns T.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(progn form \)}} \\
\text{▷ Evaluate forms sequentially. Return values of last form.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(multiple-value-prog1 form-r form \)}} \\
\text{\textbf{(multiple-value-prog2 form-r form \)}} \\
\text{▷ Evaluate forms in order. Return values\text{,primary value, respectively, of} form-r.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(tagbody \textit{tag}\textit{form} \)}} \\
\text{▷ Evaluate forms in a lexical environment. tags (symbols} \text{or integers) have lexical scope and dynamic extent, and are} \text{targets for} \textit{go}. Return NIL.} \\
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{(go \textit{tag})}} \\
\text{▷ Within the innermost possible enclosing \textit{tagbody}, jump to a tag \textit{eq\textit{tag}}.} \\
\end{align*}
\]
(⋆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 ⋆m″do")
  ▷ Evaluate ⋆tagbody-like body with vars successively bound to integers from 0 to i – 1.
  ▷ 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.

  named name
  ▷ Give ⋆loop’s implicit ⋆block a name.

  {with [(var-s)] [(var-s)]}
  ▷ [d-type] = [foo]

  {and [(var-p)] [(var-p)]}
  ▷ [d-type] = [bar]

  where destructuring type specifier d-type has the form

  {fixnum float: nil of-type}

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

  {foras [(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.

  {upfrom from downto} start
  ▷ Start stepping with start

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

  {inon} 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.

13.3 Character Syntax

#| multi-line-comment* |

: one-line-comment*
  ▷ Comments. There are stylistic conventions:

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

  ;; intro
  ▷ Description before a block of code.

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

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

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

"*
  ▷ Begin and end of a string.

'(foo)
  ▷ (quote foo); foo unevaluated.

'(foo . bar).
  ▷ 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; #On; #Xn; #Rn
  ▷ Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.

n/d
  ▷ The ratio n/d.

[m].n[([-S|D|L|E]s)] m[n][-S|D|L|E]x
  ▷ m.n*10^s as short-floating, single-floating, double-floating, long-floating, or the type from *read-default-floating-format*.

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

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

#nA
  ▷ 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-evaluated*
  ▷ 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 T] [eof-val NIL] [recursive T]])
  ▶ Return next character from stream.

(read-char-no-hang [stream [standard-input] [eof-error T] [eof-val NIL] [recursive T]])
  ▶ Next character from stream or NIL if none is available.

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

(unread-char character [stream [standard-input] [eof-error T] [eof-val NIL] [recursive T]])
  ▶ Put last read character back into stream; return NIL.

(read-byte stream [eof-error T] [eof-val NIL])
  ▶ Read next byte from binary stream.

(read-line [stream [standard-input] [eof-error T] [eof-val NIL] [recursive T]])
  ▶ Return a line of text from stream and T if line has been ended by end of file.

(read-sequence sequence stream [start start] [end end] [recursive T])
  ▶ 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])
  ▶ Copy syntax of from-char to to-readable. Return T.

*readable*
  ▶ Current readable.

*read-base* 10
  ▶ Radix for reading integers and ratios.

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

*read-suppress* T
  ▶ If T, reader is syntactically more tolerant.

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

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

(make-dispatch-macro-character char [non-term-p T] [rf] [readable])
  ▶ 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-symbol|present-symbols| external-symbol|external-symbols} {of|in} package [using (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|inconc|inconcing} {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 mloop after num iterations; num is evaluated once.

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

10 CLOS

10.1 Classes

(setq-exists-p foo bar)  ▶ T if foo has a slot bar.

(setq-boundp instance slot)  ▶ T if slot in instance is bound.

(mdefclass foo (superclass"standard-object"
  \[slot \[\{reader reader\} \[\{writer \{setf writer\}\} \[\{accessor accessor\}\] \[\{allocation \{instance \{class Instance\} \[instance\] \[\{initarg \{initarg-name\}\} \[iniform form\] \[type type\] \[documentation slot-doc\] \{\{default-initargs \{name value\}\}\] \{\{documentation class-doc\}\} \{\{metaclass name standard-class\}\}\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\]\}\]\]\]\]\]\]\]\]\]\}\]\]\]\]\]\]\}\]\]\]\]\]\]\]\]\]\]\}\]\}\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\]\}\]\}\}\]\]\}\]\]\}\]\]\}\]\]\}\]\}\}\]\}\}\]\}\}\}\]\}\}\]\}\]\}\}\]\}\}\]\}\]\}\}\]\}\}\}\}\}\}\}\}\}\]\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\}\\
(defgeneric instance new-class (initarg-slot value other-keyarg*)
  \(\triangleright\) 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 \(\text{initform}\) forms if not.

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

(update-instance-for-redefined-class new-instance added-slots
discarded-slots discarded-slots-property-list \(\{\text{initarg-slot value}\}\)
  \(\text{other-keyarg}\)*)
  \(\triangleright\) 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 \(\text{initform}\) forms. Not to be called by user.

(allocate-instance class \(\{\text{initarg-slot value}\}\)
  \(\text{other-keyarg}\)*)
  \(\triangleright\) Return uninitialized instance of class. Called by make-instance.

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

(slot-missing class instance slot \(\{\text{setf}\}
  \text{slot-bound}\)
slot-makunbound
  \(\text{slot-value}\)\)}\]
  \(\triangleright\) Called on attempted access to non-existing or unbound slot. Default methods signal error/unbound-slot, respectively. Not to be called by user.

(slot-unbound class instance slot)
  \(\triangleright\) Call on attempted access to non-existing or unbound slot. Default methods signal error/unbound-slot.

12 Types and Classes

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

(typep foo type \(\text{environment}\)*)
  \(\triangleright\) T if foo is of type.

(subtypep type-a type-b \(\text{environment}\)*)
  \(\triangleright\) Return T if type-a is a recognizable subtype of type-b, and NIL if the relationship could not be determined.

(the type form)
  \(\triangleright\) Declare values of form to be of type.

(coerce object type)
  \(\triangleright\) Coerce object into type.

(typep foo \(\text{type-a form}\)\(*\)
  \(\text{otherwise}\) \(\text{b-form}\)\(*\))
  \(\triangleright\) Return values of the first form* whose type is foo of. Return values of b-forms if no type matches.

(typep foo \(\text{type-a form}\)\(*\))
  \(\triangleright\) Return values of the first form* whose type is foo of. Signal non-convertable/convertable type-error if no type matches.

(typep foo \(\text{type importance}\)
  \(\triangleright\) Type of foo.

(check-type place type \(\text{string}\) \(\text{lambda}\) type)
  \(\triangleright\) Signal correctable type-error if place is not of type. Return NIL.

(stream-element-type stream)
  \(\triangleright\) Type of stream objects.

(array-element-type array)
  \(\triangleright\) Element type array can hold.

(upgraded-array-element-type type \(\text{environment}\))
  \(\triangleright\) Element type of most specialized array capable of holding elements of type.
Return values of forms unless restart is called during their evaluation. In this case, describe restart using \texttt{format control} and \texttt{args} (see page 38) and return \texttt{NIL}.

\begin{verbatim}
(*with-simple-restart
  (restart \{ form \})
\end{verbatim}

\texttt{format control} and \texttt{args} (see page 38) and return \texttt{NIL}.

\begin{verbatim}
(*restart-case form (restart (ord-\lambda*)
  \textbf{interactive arg-function}
  \textbf{report report-function}
  \textbf{test test-function}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} or, if during evaluation of \texttt{form} one of the dynamically established \texttt{restarts} is called, the values of its \texttt{restarts}.

\begin{verbatim}
(*restart bind
  (restart \{ form \})
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} should perform a non-local transfer of control. A restart is visible under \texttt{condition} if \texttt{test-function} \texttt{condition} \texttt{returns} \texttt{T}. If presented in the debugger, \texttt{restarts} are described by \texttt{string} or by \texttt{\#\textemdash\texttt{report-function}} (of a stream). A restart can be called by \texttt{(invoke-restart restart arg*)}, where \texttt{args} match \texttt{ord-\lambda*}, or by \texttt{(invoke-restart-interactively restart)} where a list of the respective \texttt{args} is supplied by \texttt{\#\textemdash\texttt{arg-function}}. See page 18 for \texttt{ord-\lambda*}.

\begin{verbatim}
(*invoke-restart restart arg*)
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{arg*}.

\begin{verbatim}
(*invoke-restart-interactively restart*)
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{arg*}.

\begin{verbatim}
(*find-restart
  \textbf{compute-restarts name} \{ condition \}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{name} \texttt{condition} \texttt{returns} \texttt{T}. If presented in the debugger, \texttt{restarts} are described by \texttt{restart-function} (of a stream). A restart can be called by \texttt{(invoke-restart restart arg*)}, where \texttt{args} must be suitable for the corresponding \texttt{restart-function}, or by \texttt{(invoke-restart-interactively restart)} where a list of the respective \texttt{args} is supplied by \texttt{arg-function}.

\begin{verbatim}
(*find-restart
  \textbf{compute-restarts name} \{ condition \}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{name} \texttt{condition} \texttt{returns} \texttt{T}. If presented in the debugger, \texttt{restarts} are described by \texttt{restart-function} (of a stream). A restart can be called by \texttt{(invoke-restart restart arg*)}, where \texttt{args} must be suitable for the corresponding \texttt{restart-function}, or by \texttt{(invoke-restart-interactively restart)} where a list of the respective \texttt{args} is supplied by \texttt{arg-function}.

\begin{verbatim}
(*compute-restarts name \{ condition \}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{name} \texttt{condition} \texttt{returns} \texttt{T}. If presented in the debugger, \texttt{restarts} are described by \texttt{restart-function} (of a stream). A restart can be called by \texttt{(invoke-restart restart arg*)}, where \texttt{args} must be suitable for the corresponding \texttt{restart-function}, or by \texttt{(invoke-restart-interactively restart)} where a list of the respective \texttt{args} is supplied by \texttt{arg-function}.

\begin{verbatim}
(*abortion
  \textbf{muffle-warning}
  \textbf{continue}
  \textbf{store-value value}
  \textbf{use-value value}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.

\begin{verbatim}
(*call-next-method arg*
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.

\begin{verbatim}
(*no-applicable-method generic-function arg*)
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.

\begin{verbatim}
(*invalid-method-error method
  \textbf{method-combination-error}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.

\begin{verbatim}
(*no-next-method generic-function method arg*)
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.

\begin{verbatim}
(*function-keywords method*  
  \textbf{method-qualifiers method}
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.

\begin{verbatim}
(*method-qualifiers method*)
\end{verbatim}

\texttt{elseif} \texttt{method} \texttt{function} \texttt{condition} \texttt{NIL} or \texttt{error}.
10.3 Method Combination Types

Standard

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

Examples of method-combination types: have the same usage as the c-types defined by the short form of defmethod-combination.

(defun def-method-combination c-type
  \{\begin{align*}
    \&\text{documentation} & \text{string} \\
    \&\text{identifier-with-one-argument} & \text{book}
  \end{align*}\}
  
  \text{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 method :call-next-method or from the generic function, respectively, the values of \((\text{operator (primary-method gen-arg)}))\), gen-arg* being the arguments of the generic function. The primary-methods are ordered \{(most-specific-first) \{most-specific-last\} \} (specified as c-arg in defgeneric). Using c-type as the qualifier in defmethod makes the method primary.\}

(defun def-method-combination c-type (ord-\*\*) \{(group
  \{\begin{align*}
    \&\text{qualifier}\* & \text{[]} \\
    \&\text{predicate} & \text{control}
  \end{align*}\}
  
  \text{Long Form. Define new method-combination c-type. A call to a generic function using c-type will be equivalent to a call to the forms returned by body* with ord-\*\* bound to c-arg* (cf. defgeneric), with symbol bound to the generic function, with method-combination-\* bound to the arguments of the generic function, and with groups bound to lists of methods. An applicable method becomes a member of the least-most group whose predicate or qualifiers match. Methods can be called via call-method. Lambda lists (ord-\*\*) and (method-combination-\*\*) according to ord-\* on page 18, the latter enhanced by an optional &whole argument.\}

(defun call-method
  \{(method \{(make-method-form\}) \{\begin{align*}
    \&\text{next-method} & \text{[]} \\
    \&\text{make-method-form} & \text{[]} \end{align*}\}\})
  
  \text{From within an effective method form, call method with the arguments of the generic function and with information about its next-methods; return its values.}\}

11 Conditions and Errors

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