Contents

<table>
<thead>
<tr>
<th>Numbers</th>
<th>3</th>
<th>9.5 Control Flow</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Predicates</td>
<td>3</td>
<td>9.6 Iteration</td>
</tr>
<tr>
<td>1.2</td>
<td>Numeric Funcns</td>
<td>3</td>
<td>9.7 Loop Facility</td>
</tr>
<tr>
<td>1.3</td>
<td>Logic Functions</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>Integer Functions</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>Implementation-</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dependent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Characters</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strings</td>
<td>7</td>
<td>11 Conditions and Errors</td>
<td>27</td>
</tr>
<tr>
<td>Conses</td>
<td>8</td>
<td>12 Types and Classes</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Predicates</td>
<td>8</td>
<td>13 Input/Output</td>
</tr>
<tr>
<td>4.2</td>
<td>Lists</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Association Lists</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Trees</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Sets</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Arrays</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Predicates</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Array Functions</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Vector Functions</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Sequences</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Seq. Predicates</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Seq. Functions</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Hash Tables</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structures</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Predicates</td>
<td>15</td>
<td>15 Compiler</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>Variables</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>9.3</td>
<td>Functions</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>Macros</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>CLOS</td>
<td>10</td>
<td>16 External Environment</td>
<td>47</td>
</tr>
<tr>
<td>Classes</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Types</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditions and Errors</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packages and Symbols</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packages</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbols</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Std Packages</td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Typographic Conventions

- Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.
- Placeholder for actual code.
- Literal text.
- Either one foo or nothing; defaults to bar.
- Zero or more foos.
- One or more foos.
- English plural denotes a list argument.
- Argument foo is not evaluated.
- Argument bar is possibly modified.
- foo* is evaluated as in †progn: see page 20.
- Primary, secondary, and nth return value.
- t, or truth in general; and nil or ().
1 Numbers

1.1 Predicates

\( (= \text{number} \) \)
\( (= \text{number} \) \)
\( (> \text{number} \) \)
\( (> \text{number} \) \)
\( (< \text{number} \) \)
\( (< \text{number} \) \)
\( (\text{equal} \) \)
\( (> \text{number} \) \)
\( (> \text{number} \) \)
\( (< \text{number} \) \)
\( (< \text{number} \) \)
\( (\text{minusp} \) \)
\( (\text{zerop} \) \)
\( (\text{plusp} \) \)
\( (\text{evenp} \) \)
\( (\text{oddp} \) \)
\( (\text{numberp} \) \)
\( (\text{realp} \) \)
\( (\text{rationalp} \) \)
\( (\text{floatp} \) \)
\( (\text{integerp} \) \)
\( (\text{complexp} \) \)
\( (\text{random-state-p} \) \)

1.2 Numeric Functions

\( (+ \text{a} \text{b} \) \)
\( (+ \text{a} \text{b} \) \)
\( (- \text{a} \) \)
\( (> \text{a} \) \)
\( (\text{abs} \) \)
\( (\text{floor} \) \)
\( (\text{ceiling} \) \)
\( (\text{truncate} \) \)
\( (\text{round} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{log10} \) \)
\( (\text{logbase} \) \)
\( (\text{expt} \) \)
\( (\text{exp} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
\( (\text{imag} \) \)
\( (\text{real} \) \)
\( (\text{complex} \) \)
\( (\text{conjugate} \) \)
\( (\text{imagpart} \) \)
\( (\text{realpart} \) \)
\( (\text{sqrt} \) \)
\( (\text{log} \) \)
\( (\text{sqrt} \) \)
(asinh a)  \rightarrow asinh a, \text{acosh a}, or \text{atanh a}, respectively.

\textbf{atanh a}

\textbf{cis a} \rightarrow Return \(e^{i \alpha} = \cos \alpha + i \sin \alpha\).

\textbf{conjugate a} \rightarrow Return complex conjugate of a.

\textbf{max number} + \textbf{min number} \rightarrow Greatest or least, respectively, of numbers.

\textbf{round_num} \rightarrow Return as integer or float, respectively, \(n/d\) rounded, or rounded towards \(-\infty, +\infty, 0\), or 0, respectively, and remainder.

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

\textbf{random limit} \textbf{state} \textit{random_state} \rightarrow Return non-negative random number less than \textit{limit}, and of the same type.

\textbf{make-random-state} \{state\} \textbf{ILL}: \textbf{NIL} \textbf{LIST} \rightarrow Copy of \textit{random-state} object \textit{state} or of the current random state, or a randomly initialized fresh random state.

\textbf{random-states} \rightarrow Current random state.

\textbf{float-sign number} \rightarrow \textbf{num-b} with \textbf{num-a} only.

\textbf{signum n} \rightarrow Number of magnitude 1 representing sign or phase of \textit{n}.

\textbf{numerator rational} \rightarrow Numerator or denominator, respectively, of \textit{rational}'s canonical form.

\textbf{realpart number} \rightarrow Real part or imaginary part, respectively, of \textit{number}.

\textbf{imagpart number} \rightarrow Complex real \textbf{imag}.

\textbf{phase number} \rightarrow Angle of \textit{number}'s polar representation.

\textbf{abs n} \rightarrow Return \textit{n}.

\textbf{rational real} \rightarrow \textbf{rationalize} \rightarrow Convert to real to rational. Assume complete/limited accuracy for real.

\textbf{float real [prototype]} \rightarrow Convert \textit{float} into \textit{float} with type of \textit{prototype}.

\textbf{DOUBLE-FLOAT-NUMBER-P} \rightarrow Return value of bitwise logical operation \textit{operations}.

\textbf{bool-or-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-and-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-orc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-andc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-andc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-or-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-and-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-orc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-andc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-andc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-or-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-and-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-orc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-andc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.

\textbf{bool-andc-operation} \textit{int-a} \textit{int-b} \rightarrow \textit{int-a}, \textit{int-b}.
1.4 Integer Functions

(integer-length integer) ▶ Number of bits necessary to represent integer.
(idb-test byte-spec integer) ▶ Return T if any bit specified by byte-spec in integer is set.
(ash integer count) ▶ Return copy of integer arithmetically shifted left by count adding zeros at the right, or, for count < 0, shifted right discarding bits.
(idb byte-spec integer) ▶ Extract byte denoted by byte-spec from integer. setfable.
(deposit-field (db) byte-spec integer) ▶ Return integer with bits denoted by byte-spec replaced by corresponding bits of int-a, or by the low ((byte-size byte-spec) bits of int-b, respectively.
(mask-byte-spec integer) ▶ Return copy of integer with all bits unset but those denoted by byte-spec. setfable.
(byte-size position integer) ▶ Byte specifier for a byte of size position at a starting weight of.
(byte-size byte-spec) ▶ Byte position specifier, respectively, of byte-spec.
1.5 Implementation-Dependent

- \texttt{short-float} \\
  \texttt{single-float} \\
  \texttt{double-float} \\
  \texttt{long-float} \\
  \texttt{least-negative} \\
  \texttt{least-negative-normalized} \\
  \texttt{least-positive} \\
  \texttt{least-positive-normalized} \\
  \texttt{most-negative} \\
  \texttt{most-positive} \\

- \texttt{eps} \models \text{Smallest possible number making a difference when added or subtracted, respectively.}

- \texttt{neg-eps} \models \text{from 0 current time, respectively, in clock computing precision in that radix, or}

- \texttt{curr-time} \models \text{String representing physical location of computer.}

2 Characters

- \texttt{charp} \models \text{If argument is of indicated type.}

- \texttt{standard-char-p} \models \text{If \texttt{character} is visible, alphabetic, or alphanumeric, respectively.}

- \texttt{upper-case-p} \models \text{Return \texttt{T} if \texttt{character} is uppercase, lowercase, or able to be in another case, respectively.}

- \texttt{lower-case-p} \models \text{Return \texttt{T} if \texttt{character} is a digit, or \texttt{NIL} otherwise.}

- \texttt{alpha-char-p} \models \text{Return \texttt{T} if all characters, or none, respectively, are equal.}

- \texttt{alpha-numericp} \models \text{Return \texttt{T} if \texttt{character} is uppercse, lowercase, or able to be in another case, respectively.}

- \texttt{digit-char-p} \models \text{Return \texttt{T} if all characters, or none, respectively, are equal ignoring case.}

- \texttt{char-p} \models \text{Return \texttt{T} if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.}

16 External Environment

- \texttt{get-real-time} \models \text{Current time, or computing time, respectively, in clock ticks.}

- \texttt{get-run-time} \models \text{Number of clock ticks per second.}

- \texttt{encode-universal-time} \texttt{sec min hour date month year zone} \models \text{Seconds from 1900-01-01, 00:00, ignoring leap seconds.}

- \texttt{encode-universal-time} \texttt{universal-time} \texttt{time-zone} \models \text{Return second, minute, hour, date, month, year, day, daylight-p, and zone.}

- \texttt{site-name} \models \text{String representing physical location of computer.}

- \texttt{lisp-implementation} \models \text{Name or version of implementation, operating system, or hardware, respectively.}

- \texttt{machine-instance} \models \text{Computer name.}
### 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, ignoreable, ignore, inline, notinline, optimize, or special. See below.

#### (declaration foo*)
> Make foo 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.

#### (ignoreable fvar [ignore] (function function*)]
> Suppress warnings about used/unused bindings.

### 3. Strings

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

#### (stringp foo)
> T if foo is of indicated type.

#### (simple-string-p foo)
> T if foo is of indicated type.

#### (string= foo bar)
#### (string<= foo bar)
#### (string< foo bar)
#### (string>= foo bar)
#### (string> foo bar)
> Return T if subsequences of foo and bar are equal. Obey/ignore, respectively, case.

#### (make-string size [initial-element char] [element-type type element-type*])
> Return string of length size.

#### (string x)
#### (string-capitalize x)
#### (string-upcase string)
#### (string-downcase string)
> Convert x (symbol, string, or character) into a string, a string with capitalized words, an all-upper case string, or an all-lowercase string, respectively.

#### (nstring-capitalize nstring-upcase nstring-downcase)
> Convert string into a string with capitalized words, an all-upper-case string, or an all-lowercase string, respectively.

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

#### (character c)
> Return #\c.

#### (char-lessp character+)
#### (char-not-lessp character+)
#### (char-lessp character+)
#### (char-not-greaterp character+)
> Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.

#### (char-upcase character)
#### (char-downcase character)
> Return corresponding uppercase/lowercase character, respectively.

#### (digit-char i [radix])
> Character representing digit i.

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

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

#### (char-int character)
> Code of character.

#### (char-code character)
> Character with code.

#### (char-code-limit)
> Upper bound of (char-code char); ≥ 96.
4 Conses

4.1 Predicates

\[\text{not} \quad \text{foo} \quad \Rightarrow \quad \text{Return } \text{T} \text{ if } \text{foo} \text{ is of indicated type.}\]

\[\text{cons} \quad \text{foo} \quad \Rightarrow \quad \text{Return list of } \text{foo} \text{ and NIL.}\]

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

\[\text{atom} \quad \text{foo} \quad \Rightarrow \quad \text{Return } \text{T} \text{ if } \text{foo} \text{ is not a cons.}\]

\[\text{tail} \quad \text{foo list} \quad \Rightarrow \quad \text{Return } \text{T} \text{ if } \text{foo} \text{ is a tail of list.}\]

\[\text{member} \quad \text{foo list} \quad \Rightarrow \quad \text{Return tail of list starting with its first element matching } \text{foo}. \text{ Return NIL if there is no such element.}\]

\[\text{member-if} \quad \text{foo list} \quad \Rightarrow \quad \text{Return tail of list starting with its first element satisfying } \text{test}. \text{ Return NIL if there is no such element.}\]

\[\text{subsetp} \quad \text{list-a list-b} \quad \Rightarrow \quad \text{Return } \text{T} \text{ if list-a is a subset of list-b.}\]

4.2 Lists

\[\text{cons} \quad \text{foo bar} \quad \Rightarrow \quad \text{Return new cons } \{\text{foo , bar}\}.\]

\[\text{list} \quad \text{foo} \quad \Rightarrow \quad \text{Return list of } \text{foo}.\]

\[\text{list+} \quad \text{foo} \quad \Rightarrow \quad \text{Return list of } \text{foo} \text{ with last } \text{foo} \text{ becoming cdr of last cons. Return } \text{foo} \text{ if only one } \text{foo} \text{ given}.\]

\[\text{make-list} \quad \text{number} \quad \text{initial-element} \quad \text{foo} \quad \Rightarrow \quad \text{New list with number elements set to } \text{foo}.\]

\[\text{list-length} \quad \text{list} \quad \Rightarrow \quad \text{Length of list; NIL for circular list.}\]

\[\text{car} \quad \text{list} \quad \Rightarrow \quad \text{Car of list or NIL if list is NIL. setfable.}\]

\[\text{cdr} \quad \text{list} \quad \Rightarrow \quad \text{Cdr of list or NIL if list is NIL. setfable.}\]

\[\text{rest} \quad \text{list} \quad \Rightarrow \quad \text{Cdr of list or NIL if list is NIL. setfable.}\]

\[\text{nthcdr} \quad \text{n} \quad \text{list} \quad \Rightarrow \quad \text{Return tail of list after calling } \text{cdr} \text{ n times.}\]

\[\text{first} \quad \text{second} \quad \text{third} \quad \text{fourth} \quad \text{fifth} \quad \text{sixth} \quad \ldots \quad \text{ninth} \quad \text{tenth} \quad \text{list} \quad \Rightarrow \quad \text{Return nth element of list if any, or NIL otherwise. setfable.}\]

\[\text{nth} \quad \text{n} \quad \text{list} \quad \Rightarrow \quad \text{Zero-indexed nth element of list. setfable.}\]

\[\text{cadr} \quad \text{list} \quad \Rightarrow \quad \text{With X being one to four as and ds representing } \text{cadr} \text{ and } \text{cadrs}, \text{ e.g. } (\text{cadr} \text{ bar}) \text{ is equivalent to } (\text{car} (\text{cadr} \text{ bar})). \text{ setfable.}\]

\[\text{last} \quad \text{list} \quad \text{number} \quad \Rightarrow \quad \text{Return last number conses of list.}\]
14.4 Standard Packages

commom-lisp-cl
  ➤ Exports the defined names of Common Lisp except for those in the keyword package.

commom-lisp-user-cl-user
  ➤ Current package after startup; uses package common-lisp.

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

15 Compiler

15.1 Predicates

(special-operator-p foo)
  ➤ 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
    [setf name] [definition]]
  ➤ Return compiled function or replace name's function definition with the compiled function. Return T in case of warnings or errors, and T in case of warnings or errors excluding style-warnings.

(compile-file
  [output-file out-path]
    [verbose boolean]
    [generate-partial-function boolean]
    [print boolean]
    [external-format file-format]
  ➤ Write compiled contents of file to out-path. Return true if output path or NIL, T in case of warnings or errors, T in case of warnings or errors excluding style-warnings.

(compile-file-pathname
  [output-file path] [other-keyargs]]
  ➤ Pathname, compile-file writes to if invoked with the same arguments.

(load path
  [verbose boolean]
  [generate-partial-function boolean]
  [print boolean]
  [external-format file-format]
  ➤ Load source file or compiled file into Lisp environment. Return T if successful.

4.3 Association Lists

(pairlis keys [values])
  ➤ 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
  ➤ First cons whose car, or cdr, respectively, satisfies test.

assoc-if-not key function
  ➤ Return list if foo is already member of list. If not, return (cons foo list).

assoc-if-not key function
  ➤ Return list if foo is already member of list. If not, return (cons foo list).
4.4 Trees

\(\text{tree-equal} \text{ foo} \text{ bar} \\{\text{test test expand} \}\)\n
\(\rightarrow \text{Return } \text{T} \text{ if trees } \text{foo} \text{ and } \text{bar} \text{ have same shape and leaves satisfying test.}\)

\(\{\text{subst new old tree} \}\) \(\{\text{test function expand} \}\)

\(\rightarrow \text{Make copy of } \text{tree} \text{ with each subtree or leaf matching } \text{old} \text{ replaced by } \text{new.}\)

\(\{\text{subst-if-not old new test tree} \}\) \(\{\text{test function expand} \}\)

\(\rightarrow \text{Make copy of } \text{tree} \text{ with each subtree or leaf satisfying } \text{test} \text{ replaced by } \text{new.}\)

\(\{\text{sublis association-list tree} \}\) \(\{\text{test function expand} \}\)

\(\rightarrow \text{Make copy of } \text{tree} \text{ with each subtree or leaf matching a key in } \text{association-list} \text{ replaced by that key’s value.}\)

\(\{\text{copy-tree tree} \}\) \(\rightarrow \text{Copy of } \text{tree} \text{ with same shape and leaves.}\)

4.5 Sets

\(\{\text{intersection } \}\)

\(\{\text{set-difference } \}\)

\(\{\text{union } \}\)

\(\{\text{set-exclusive-or } \}\)

\(\{\text{nintersection } \}\)

\(\{\text{nset-difference } \}\)

\(\{\text{nunion } \}\)

\(\{\text{nset-exclusive-or } \}\)

\(\rightarrow \text{Return } a \cap b, a \setminus b, a \cup b, \text{ or } a \triangle b, \text{ respectively, of lists } a \text{ and } b.\)

5 Arrays

5.1 Predicates

\(\{\text{arrayp foo} \}\)

\(\{\text{vectorp foo} \}\)

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

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

\(\{\text{simple-bit-vector-p foo} \}\)

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

\(\{\text{array-has-fill-pointer-p array} \}\)

\(\rightarrow \text{T if array is adjustable/has a fill pointer, respectively.}\)

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

\(\rightarrow \text{T if subscripts are in array's bounds.}\)

5.2 Array Functions

\(\{\text{make-array dimension-sizes }\text{[adjustable boole]} \}\)

\(\{\text{adjust-array dimension-sizes }\text{[element-type type]} \}\)

\(\{\text{fill-pointer }\text{[num boole]} \}\)

\(\{\text{initial-element obj} \}\)

\(\{\text{initial-contents tree-or-array} \}\)

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

\(\rightarrow \text{Return fresh, or readjust, respectively, vector or array.}\)

\(\{\text{aref array }\text{[subscripts]}\}\)

\(\rightarrow \text{Return array element pointed to by subscripts. }\text{setfable.}\)

\(\{\text{row-major-aref array }\text{i }\}\)

\(\rightarrow \text{Return } \text{i} \text{th element of array in row-major order. }\text{setfable.}\)

14.3 Symbols

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

\(\{\text{make-symbol name}\}\)

\(\rightarrow \text{Make fresh, uninterned symbol name.}\)

\(\{\text{gensym}\}\)

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

\(\{\text{gentemp}\text{[prefix package]}\}\)

\(\rightarrow \text{Intern fresh symbol in package. Depreciated.}\)

\(\{\text{copy-symbol symbol }\text{[props]}\}\)

\(\rightarrow \text{Return uninterned copy of symbol. If } \text{props} \text{ is } \text{T}, \text{ give copy the same value, function and property list.}\)
14 Packages and Symbols

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

14.1 Predicates

(symbolp foo)  ▶ If foo is of indicated type.
(packagep foo)  ▶ Packages used by/using foo.
(keywordp foo)  ▶ Common Lisp-user dimensions of foo.

14.2 Packages

:bar|keyword:bar ▶ Keyword, evaluates to :bar.
package:symbol ▶ Exported symbol of package.
package:~symbol ▶ Possibly unexported symbol of package.

(defun package-use-list (package)
  "List of other packages used by/using package."
  (values)
)

(defun delete-package (package)
  "Delete package. Return T if successful."
  (values)
)

(defun *packages (common-lisp-user)
  "The current package."
  (values)
)

(defun list-all-packages
  "List of registered packages."
  (values)
)

(defun package-name (package)
  "Name of package."
  (values)
)

(defun package-nicknames (package)
  "Nicknames of package."
  (values)
)

(defun find-package (name)
  "Package with name (case-sensitive)."
  (values)
)

(defun find-all-symbols (foo)
  "List of symbols foo from all registered packages."
  (values)
)

(defun make-package (package)
  "Create or modify package with interned-symbols, symbols from used-packages, imported-symbols, and shd-symbols. Add shd-symbols to foo's shadowing list."
  (values)
)

(defun rename-package (package new-name)
  "Rename package. Return renamed package."
  (values)
)

(defun use-package other-packages (package)
  "Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T."
  (values)
)

(defun use-package (other-packages package)
  "Use other-packages in package."
  (values)
)

(defun *use-package (other-packages package)
  "Use other-packages in package."
  (values)
)

(defun *package-use-list package)
  "List of other packages used by/using package."
  (values)
)

(defun *delete-package package)
  "Delete package. Return T if successful."
  (values)
)

(defun *packages common-lisp-user
  "The current package."
  (values)
)

(defun list-all-packages
  "List of registered packages."
  (values)
)

(defun package-name package
  "Name of package."
  (values)
)

(defun package-nicknames package
  "Nicknames of package."
  (values)
)

(defun find-package name
  "Package with name (case-sensitive)."
  (values)
)

(defun find-all-symbols foo
  "List of symbols foo from all registered packages."
  (values)
)

(defun array-row-major-index array (subscripts)
  "Index in row-major order of the element denoted by subscripts."
  (values)
)

(defun array-dimensions array
  "List containing the lengths of array's dimensions."
  (values)
)

(defun array-dimension array i
  "Length of ith dimension of array."
  (values)
)

(defun array-total-size array
  "Number of elements in array."
  (values)
)

(defun array-rank array
  "Number of dimensions of array."
  (values)
)

(defun array-displacement array target-array offset
  "Target array and offset."
  (values)
)

(defun bit bit-array (subscripts)
  "Return element of bit-array or of simple-bit-array. setfable."
  (values)
)

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

(defun array-rank-limit
  "Upper bound of array rank; ≥ 8."
  (values)
)

(defun array-dimension-limit
  "Upper bound of an array dimension; ≥ 1024."
  (values)
)

(defun array-total-size-limit
  "Upper bound of array size; ≥ 1024."
  (values)
)

5.3 Vector Functions

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

(defun vector foo*)  "Return fresh simple vector of foos."

(defun sref vector i
  "Element i of simple vector. setfable."
  (values)
)

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

(defun vector-push-extend foo vector num
  "Replace element of vector pointed to by fill pointer with foo, then increment fill pointer. Extend vector’s size by num if necessary."
  (values)
)

(defun vector-pop vector
  "Return element of vector its fillpointer points to after decrementation."
  (values)
)

(defun fill-pointer vector
  "Fill pointer of vector. setfable."
  (values)
6 Sequences

6.1 Sequence Predicates

\(\{\text{every} / \text{notevery}\} \text{ test sequence}^+\)

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

\(\{\text{some} / \text{notany}\} \text{ test sequence}^+\)

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

\(\text{\mismatch sequence-a sequence-b} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Return position in \(\text{sequence-a}\) where \(\text{sequence-a}\) and \(\text{sequence-b}\) begin to mismatch. Return \(\text{NIL}\) if they match entirely.

6.2 Sequence Functions

\(\text{\make-sequence sequence-type size \{initial-element\} foo}\)

\(\text{\(\Rightarrow\)}\) Make sequence of sequence-type with size elements.

\(\text{\concatenate type sequence}\)

\(\text{\(\Rightarrow\)}\) Return concatenated sequence of type.

\(\text{\merge type sequence-a sequence-b test \{key function\} body}\)

\(\text{\(\Rightarrow\)}\) Return interleaved sequence of type. Merged sequence will be sorted if both \(\text{sequence-a}\) and \(\text{sequence-b}\) are sorted.

\(\text{\fill sequence foo \{start\} \{end\} \{start1\} \{end1\} \{start2\} \{end2\} \{key function\}}\)

\(\text{\(\Rightarrow\)}\) Return sequence after setting elements between \(\text{start}\) and \(\text{end}\) to \(\text{foo}\).

\(\text{\length sequence} \text{ function}\)

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

\(\text{\{count \{count-if\} \{count-if-not\}\} test sequence \{from-end\} \{test\ function\} \{test-not\ function\} \{start\} \{end\} \{key function\}}\)

\(\text{\(\Rightarrow\)}\) Return number of elements in sequence which match \(\text{foo}\).

\(\text{\elt sequence index} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Return element of \(\text{sequence}\) pointed to by zero-indexed index. setable.

\(\text{\subsetseq sequence start \{end\}}\)

\(\text{\(\Rightarrow\)}\) Return subsequence of sequence between \(\text{start}\) and \(\text{end}\). setable.

\(\text{\sort sequence test \{key function\}}\)

\(\text{\(\Rightarrow\)}\) Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

\(\text{\reverse sequence} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Return \(\text{sequence}\) in reverse order.

\(\text{\nreverse sequence} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Return \(\text{sequence}\) in reverse order.

\(\text{\parse-namestring foo host \{defalut-pathname\}}\)

\(\text{\(\Rightarrow\)}\) Return path name converted from string, path name, or stream \(\text{foo}\) and position where parsing stopped.

\(\text{\merge-pathnames path-or-stream \{path-name\}}\)

\(\text{\(\Rightarrow\)}\) Return minimal path string that sufficiently describes the path of path-or-stream relative to root-path.

\(\text{\nombrex path-or-stream \{path-name\}}\)

\(\text{\(\Rightarrow\)}\) Return string representing full path name: name, type, and version; directory name; or host name, respectively, of path-or-stream.

\(\text{\translate-pathname path-or-stream wildcard-path-a \{default\}}\)

\(\text{\(\Rightarrow\)}\) Translate the path of path-or-stream from wildcard-path-a into wildcard-path-b. Return new path.

\(\text{\pathname path-or-stream} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Pathname of path-or-stream.

\(\text{\logical-pathname logical-path-or-stream} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Logical pathname of logical-path-or-stream. Logical pathnames are represented as all-uppercase namestring.

\(\text{\logical-pathname-translations logical-host} \text{ function}\)

\(\text{\(\Rightarrow\)}\) List of (from-wildcard to-wildcard) translations for logical-host. setable.

\(\text{\load-logical-pathname-translations logical-host} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Load logical-host’s translations. Return \(\text{NIL}\) if already loaded; return \(\text{T}\) if successful.

\(\text{\translate-logical-pathname path-or-stream} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Physical pathname corresponding to (possibly logical) pathname of path-or-stream.

\(\text{\probe-file file} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Canonical name of file. If file does not exist, return \(\text{NIL}/\text{signal file-error}\), respectively.

\(\text{\file-write-date file} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Time at which file was last written.

\(\text{\file-author file} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Return name of file owner.

\(\text{\file-length stream} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Return length of stream.

\(\text{\rename-file foo bar} \text{ function}\)

\(\text{\(\Rightarrow\)}\) 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.

\(\text{\delete-file file} \text{ function}\)

\(\text{\(\Rightarrow\)}\) Delete file. Return \(\text{T}\).

\(\text{\directory path} \text{ function}\)

\(\text{\(\Rightarrow\)}\) List of pathnames matching path.
13.7 Pathnames and Files

(makepathname
  [:host {host NIL unspecific}
   :device {device NIL unspecific}
   :directory {directory {absolute NIL wildcard}
     {relative NIL wildcard}
     :back NIL unspecific}
   :name {file-name {wild NIL wildcard}
     :up NIL unspecific}
   :type {file-type {wild NIL wildcard}
     :version {newest version {wild NIL wildcard}
       :defaults path
       :case {local common} common}
     :case {local common} local]
  ])

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

(pathname-host
  /pathname-device
  (/pathname-directory path-or-stream [:case {local common} common]
  /pathname-name
  /pathname-type
  /pathname-version path-or-stream]
)

> Return pathname component.
7 Hash Tables

The Loop Facility provides additional hash-table-related functionality; see loop, page 21.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

(replace sequence-a sequence-b)
  ▶ Replace elements of sequence-a with elements of sequence-b.

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

(map-into result-sequence function sequence+)
  ▶ Store into result-sequence successively values of function applied to corresponding elements of the sequences.

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

(cop-seq sequence)
  ▶ Copy of sequence with shared elements.

13.6 Streams

(make-concatenated-stream input-stream+)
  ▶ Return a stream of indicated type.

(make-string-input-stream string [start [end]])
  ▶ 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.

(make-two-way-stream input-stream-part output-stream-part)
  ▶ Return two-way stream from-input-stream to-output-stream.

(make-synonym-stream variable-bound-to-stream)
  ▶ Return synonym stream.

(make-hash-table (size int) (rehash-size num) (rehash-threshold num)
  ▶ Make a hash table.

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

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

(with-hash-table-test (foo hash-table) (declare (fixnum) ... form)
  ▶ Return values of forms. In forms, invocations of (foo) return T if an entry is returned; its key; its value.

(map-hash-table test hash-table)
  ▶ Test function used in hash-table.

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

(sxhash foo)
  ▶ Hash code unique for any argument, equal foo.
9.1 Predicates

- (copy-structure structure) Return copy of structure with shared slot values.
- (type-definition class) (type-definition type) (type-definition keyword) is a special form.
- (concrete-class name) (concrete-class keyword) (concrete-class keyword) is a special form.
- (type vector type) (type vector keyword) (type vector keyword) is a special form.
- (make-instance constructorargs) (make-instance constructorargs) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
- (type equality predicate) (type equality predicate) is a special form.
9.2 Variables

(defun constant (foo form (doc))
  \( \Rightarrow \) Assign value of form to global constant/dynamic variable foo.

(defun parameter (foo form (doc))
  \( \Rightarrow \) Unless bound already, assign value of form to dynamic variable foo.

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

(set \( \Rightarrow \) symbol form)
  \( \Rightarrow \) Set symbol’s value cell to form. Deprecated.

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

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

(rotatel \( \Rightarrow \) place+)*
  \( \Rightarrow \) Rotate values of places left, old first becoming new last place’s value. Return nil.

(makunbound foo)
  \( \Rightarrow \) Delete special variable foo if any.

(get symbol key \( \Rightarrow \) \( \text{default nil} \))

(getf place key \( \Rightarrow \) \( \text{default nil} \))
  \( \Rightarrow \) First entry key from property list stored in symbol/in place, respectively, or default if there is no key. setfable.

(get-properties property-list keys)
  \( \Rightarrow \) 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.

(rempop symbol key)

(rempf place key)
  \( \Rightarrow \) 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*)
  \( \Rightarrow \) Evaluate forms with locally established dynamic bindings of symbols to values or nil. Return values of forms.

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

(multiple-value-bind \( \Rightarrow \) \( \text{var} \) values form (declare \( \text{decl*} \) form*)
  \( \Rightarrow \) Evaluate body-forms with vars lexically bound to the return values of values-form. Return values of body-forms.

- [min-col \( \Rightarrow \) [col-in \( \Rightarrow \) ] [min-pad \( \Rightarrow \) ] [pad-char \( \Rightarrow \) ] ]
  \( \Rightarrow \) Aesthetic/Standard. Print argument of any type for consumption by humans/by the reader, respectively. With ;, print nil as () rather than nil; with @, add pad-char on the left rather than on the right.

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

- [R \( \Rightarrow \) R | -R | @R | @R]
  \( \Rightarrow \) Roman. Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

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

- [width \( \Rightarrow \) [dec-digits \( \Rightarrow \) ] [shift \( \Rightarrow \) ] [overflow-char \( \Rightarrow \) ] [pad-char \( \Rightarrow \) ] ] \text{go} \text{r}]
  \( \Rightarrow \) Fixed-Format Floating-Point. With @, always prepend a sign.

- [width \( \Rightarrow \) [dec-digits \( \Rightarrow \) ] [exp-digits \( \Rightarrow \) ] [scale-factor \( \Rightarrow \) ] [overflow-char \( \Rightarrow \) ] [pad-char \( \Rightarrow \) ] ] \text{go} \text{r}]
  \( \Rightarrow \) Exponential/General Floating-Point. Print argument as floating-point number with dec-digits after decimal point and exp-digits in the signed exponent. With @, choose either -e or -f. With @, always prepend a sign.

- [dec-digits \( \Rightarrow \) [int-digits \( \Rightarrow \) ] [width \( \Rightarrow \) [pad-char \( \Rightarrow \) ] ] \text{go} \text{r}]
  \( \Rightarrow \) Monetary Floating-Point. Print argument as fixed-format floating-point number. With ;, put sign before any padding; with @, always prepend a sign.

- [\text{c} | -\text{c} | \text{oc} | @\text{c}] \( \Rightarrow \) Character. Print, spell out, print in #\# syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

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

- [p \( \Rightarrow \text{p} \) | @p | @p]
  \( \Rightarrow \) Plural. If argument eq 1 print nothing, otherwise print a, do the same for the previous argument; if argument eq 1 print y, otherwise print yes; do the same for the previous argument, respectively.

- [\text{e} | %]
  \( \Rightarrow \) Newline. Print n newlines.

- [@ | &]
  \( \Rightarrow \) Fresh-Line. Print n – 1 newlines if output stream is at the beginning of a line, or n newlines otherwise.

- [\text{c} | -\text{c} | @\text{c} | @\text{c}]
  \( \Rightarrow \) Conditional Newline. Print a newline like print-newline with argument :linear, :fill, :mirror, or :mandatory, respectively.

- [\text{e} | @e | @e]
  \( \Rightarrow \) Ignored Newline. Ignore newline, or whitespace following newline, or both, respectively.

- [@ | @]
  \( \Rightarrow \) Page. Print n page separators.

- [@ | @]
  \( \Rightarrow \) Tilde. Print n tildes.

- [\text{min-col} \( \Rightarrow \) [col-in \( \Rightarrow \) ] [min-pad \( \Rightarrow \) ] [pad-char \( \Rightarrow \) ] ]
  \( \Rightarrow \) nl-text \( \Rightarrow \) \text{span} \[ \text{width} \] \{ text \( \Rightarrow \) \} text
(print-newline :stream :linear :fill :mandatory)  
▷ 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.

*print-case*  
▷ Print symbol names all uppercase (:upcase), all lowercase (:downcase), or 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*  

*print-level*  

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

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

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

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

13.5 Format

(format-control (stream and arg*) applying format to stream, control, and arg* returning NIL or any excess args.

(format (stream out-string out-stream) control arg*)  
▷ Output string control which may contain - directives possibly taking some args. Alternatively, control can be a function returned by *format 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.

(form-destructuring-bind destruct-λ bar (declare decr* form))  
▷ Evaluate forms with variables from tree destruct-λ bound to corresponding elements of tree bar, and return their values. destruct-λ resembles macro-λ (section 9.4), but without any &environment clause.

9.3 Functions

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

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

\[\begin{align*} \&\text{key} & \{\text{var} \&\text{key} \&\text{rest} \{\text{var} \&\text{init} \&\text{supplied-}\})^* \} \\
\&\text{allow-other-keys} & \{\&\text{aux} \&\text{var} \&\text{init} \&\text{supplied-}\}^* \\
\end{align*}\]

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

{\{defun foo (ord-λ*) \{setf foo \{new-value ord-λ*\}) (declare decr* \{form\}) \{doc \{form\}\} \\
▷ Define a function named foo or (setf foo), or an anonymous function, respectively, which applies forms to ord-λs. For defun, forms are enclosed in an implicit \textit{block} named foo.

\{\{let \{labels\} \{\{foo ord-λ*\} \{setf foo \{new-value ord-λ*\}) (declare local-decr*) \{doc \{local-form\} \{declare decr*\}) \{form\} \\
▷ Evaluate forms with locally defined functions foo. Globally defined functions of the same name are shadowed. Each foo is also the name of an implicit \textit{block} around its corresponding local-forms. Only for \{labels\}, functions foo are visible inside local-forms. Return values of forms.

\{function \{foo \{\{lambda \{form\}\}\}_\{\text{life}\}\} \\
▷ Return function of stream and arg* applying \textit{format} to stream, control, and arg* returning NIL or any excess args.

\{format \{\{\text{string} \{\text{control} \{\text{arg}\}^*\}\} \{\text{arg}\}^*\} \{\{\text{arg}\}^*\}
▷ Call function with all the values of each form as its arguments. Return values returned by function.

\{values-list \{list\}\}  
▷ Return elements of \{list\}.

\{values \{foo\} \{\text{life}\}\}  
▷ Return as multiple values the primary values of the foos. settable.

\{multiple-value-list \{form\}\}  
▷ List of the values of \{form\}.

\{\text{mth-value} \{n\} \{form\}\}  
▷ Zero-indexed \text{mth} return value of \{form\}.

\{complement \{function\}\}  
▷ Return new function with same arguments and same side effects as \{function\}, but with complementary truth value.

\{constantly \{foo\}\}  
▷ Function of any number of arguments returning foo.

\{identity \{foo\}\}  
▷ Return \{foo\}.\]
9.4 Macros

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

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

\[(\text{\&defmacro })(\text{\&define-complier-macro })(\text{\&define-symbol-macro })(\text{\&defsetf})\]


---

(*write-char* \text{char} \text{stream})
\> Output \text{char} to \text{stream}.

(*write-string* \text{string} \text{stream})
\> Write \text{string} to \text{stream} without/with a trailing newline.

(*write-byte* \text{byte} \text{stream})
\> Write byte to binary stream.

(*write-sequence* \text{sequence} \text{stream})
\> Write elements of \text{sequence} to binary or character stream.

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

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

\[(\text{\&print-logical-block} \text{\&stream} \text{\&prefix string} \text{\&suffix string})\]
\> Evaluate forms, which should print \text{list}, with \text{stream} locally bound to a pretty printing stream which outputs to the original \text{stream}. If \text{list} is in fact not a list, it is printed by /write. Return \text{NIL}.

\[(\text{\&print-pop})\]
\> Take next element off \text{list}. If there is no remaining tail of \text{list}, or \text{\&print-lengths} or \text{\&print-circles} indicate printing should end, send element together with an appropriate indicator to \text{stream}.

\[(\text{\&print-tab} \text{\&line-relate} \text{\&section} \text{\&section-relative} \text{\&circle})\]
\> Move cursor forward to column number \(c + k\), \(k \geq 0\) being as small as possible.

\[(\text{\&print-indent} \text{\&block} \text{\&section} \text{\&section-relative} \text{\&block})\]
\> Specify indentation for innermost logical block relative to leftmost position/to current position. Return \text{NIL}.

\[(\text{\&print-put-if-list-exhausted})\]
\> If \text{list} is empty, terminate logical block. Return \text{NIL} otherwise.
n/d ▶ The ratio \( \frac{a}{b} \).

\[
\{\text{func} \mid (\text{type} \{\text{slot value}\})\} \ni \{\text{value} \mid (\text{type} \{\text{slot value}\})\}
\]

\( m \cdot n \cdot 10^8 \text{ as short-float, single-float, double-float, long-float, or the type from } \text{read-default-float-format}.\)

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

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

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

#[n]*b* ▶ Bit vector of some \(n\) \text{b} filled with last \text{b} if necessary.

\(\text{S}(\text{type} \{\text{slot value}\})\) ▶ Structure of type.

#P(string) ▶ A pathname.

#:foo ▶ Uninterned symbol \text{foo}.

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

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

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

#:integer# ▶ Object labelled \text{integer}.

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

#:feature when-feature ▶ Means when-feature if \text{feature} is \text{T}; means unless-feature if \text{feature} is \text{NIL}. \text{feature} is a symbol from \text{*,features}, or \((\text{and/or} \text{feature})*\), or \(\text{not feature}\).

*,\text{features} ▶ List of symbols denoting implementation-dependent features.

\(c^*\); \(\backslash c\) ▶ Treat arbitrary character(s) \(c\) as alphabetic preserving case.

13.4 Printer

\(\backslash\text{print1}\) \(\backslash\text{print}\) \(\backslash\text{prin}\) \(\backslash\text{prin1}\)

▪ Print \text{foo} to \text{stream}, \text{readably}, \text{readably} between a new-line and a space, \text{readably} after a new-line, or human-readable without any additional characters, respectively. \text{print1}, \text{prin}, \text{prin1} return \text{foo}.

\(\backslash\text{prin1-to-string}\) \(\backslash\text{prin-to-string}\)

▪ Print \text{foo} to \text{string}, \text{readably} or human-readable, respectively.

\(\backslash\text{print-object}\) \(\backslash\text{print-unreadable-object}\)

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

\(\backslash\text{print-unreadable-object}\)

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

\(\backslash\text{terpri}\)

▪ Output a new-line to \text{stream}. Return \text{NIL}.

\(\backslash\text{fresh-line}\)

▪ Output a new-line to \text{stream} and return \text{T} unless \text{stream} is already at the start of a line.

9.5 Control Flow

\(\backslash\text{if}\) \(\backslash\text{then}\) \(\backslash\text{else}\)

▪ Return values of \text{then} if \text{test} returns \text{T}; return values of \text{else} otherwise.

\(\backslash\text{cond}\)

▪ Return the values of the first \text{then} whose \text{test} returns \text{T}; return \text{NIL} if all tests return \text{NIL}.

\(\backslash\text{when}\) \(\backslash\text{unless}\)

▪ Evaluate \text{foo}s and return their values if \text{test} returns \text{T} or \text{NIL}, respectively. Return \text{NIL} otherwise.
(n-case test ((key) *foo*) ((otherwise) bar *end))
> Return the values of the first foo* one of whose keys is eql test. Return values of bar if there is no matching key.

(n-case test ((key) *foo*) (key))
> 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.

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

(or form)*
> Evaluate forms from left to right. Immediately return 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*)
> Evaluate forms in order. Return values/primary value, respectively, of form-r.

(tagbody 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])
> 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, eq 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.

(\read-sequence sequence stream [start start] [end end])
> 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 new)\readtable
> Case sensitivity attribute (one of upcase, downcase, preserve, invert) of readable, settable.

(\copy-readable from-readable new)\readtable
> Return copy of from-readable.

(\set-syntax-from-char to-char from-char \readtable)
> Set syntax of from-char to to-readable. Return T.

(\readables)
> Current readable.

(\read-base)
> 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)
> If T, reader is syntactically more tolerant.

(\set-macro-character char function \readtable)
> Make char a macro character associated with function of stream and char. Return T.

(\get-macro-character char \readtable)
> Reader macro function associated with char, and T if char is a non-terminating macro character.

(\make-dispatch-macro-character char \readtable)
> Make char a dispatching macro character. Return T.

(\set-dispatch-macro-character char sub-char function \readtable)
> Make function of stream, n, sub-char a dispatch function of char followed by n, followed by sub-char. Return T.

(\get-dispatch-macro-character char sub-char \readtable)
> Dispatch function associated with char followed by sub-char.

13.3 Character Syntax

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

;ofilte
> Short title for a block of code.

; intro
> Description before a block of code.

; state
> State of program or of following code.

; explanation
> Regarding line on which it appears.

; continuation
> Begin and end of a string.

'foo
> (quote foo); foo unevaluated.

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

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

#Bn; #On; n.; #Xn; \#r\Rn
> Integer of radix 2, 8, 10, 16, or \r; 2 \leq r \leq 36.
13 Input/Output

13.1 Predicates

- `stream-p foo` → T if foo is of indicated type.
- `pathname-p foo` → T if foo is of indicated type.
- `readable-p foo` → T if foo is of indicated type.
- `input-stream-p stream` → Return T if stream is for input, for output, interactive, or open, respectively.
- `output-stream-p stream` → Return T if stream is for input, for output, interactive, or open, respectively.
- `interactive-stream-p stream` → Return T if stream is for input, for output, interactive, or open, respectively.
- `open-stream-p stream` → Return T if stream is for input, for output, interactive, or open, respectively.
- `pathname-match-p path wildcard` → T if path matches wildcard.
- `wild-pathname-p path [[:host[:device:directory:]name:][:type:version:][:NIL:]]` → Return T if indicated component in path is wildcard. (NIL indicates any component.)

13.2 Reader

- `{y-or-n-p | yes-or-no-p} {control arg*}` → Ask user a question and return T or NIL depending on their answer. See page 36, format. For control and args.
- `read` → Read list of characters from standard input as S-expression.
- `read-preserving-whitespace` → Read list of characters from standard input as S-expression preserving whitespace.
- `read-from-string string [eof-err |eof-value |recursive]` → Return list of objects read. Signal error if no char is found in string.
- `read-char [stream] [standard-input] [eof-err |eof-value |recursive]` → Return next character from stream.
- `read-char-no-hang [stream] [standard-input] [eof-err |eof-value |recursive]` → Return next character from stream or NIL if none is available.
- `peek-char [mode] [stream] [standard-input] [eof-err |eof-value |recursive]` → Next, or if mode is 1, 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]` → Put last `read-char` character back into stream; return NIL.
- `read-byte stream [eof-err] [eof-value]` → Read next byte from binary stream.
- `read-line [stream] [standard-input] [eof-err] [eof-value]` → Return a line of text from stream and T if line has been ended by end of file.

9.6 Iteration

- `{do (var start [step]) (stop result) (declare 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.
- `{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 decl` → Give `loop`'s implicit `block` a name.

- `{with (var-s [var-s] | [d-type] | [= foo]|)` → Destructuring specifier.
- `{and (var-p [var-p]) [d-type] | [= bar]|)` → Destructuring specifier.

- `{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} [downto] [to] [below] [above] form` → Specify form as the end value for stepping.
- `{in} [on] list` → Bind var to successive elements/tails, respectively, of list.
- `{step} [function] [on]` → Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.
- `{hash-key} [hash-keys] [on] hash-table [using] [hash-value]` → Bind var successively to the keys of `hash-table`; bind value to corresponding values.
12 Types and Classes

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

(typep foo (environment))  → T if foo is of type.

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

(the type form)  → Declare values of form to be of type.

(coerce object type)  → Coerce object into type.

(typecase foo (type a-form)*) [ [ (otherwise) b-forms ]]  → Return values of the first a-form* whose type is foo of. Return values of b-forms if no type matches.

(metypecase) foo (type b-form)*  → Return values of the first form* whose type is foo of. Signal non-correctable/correctable type-error if no type matches.

(type-of foo)  → Type of foo.

(check-type place type [string])  → Signal correctable type-error if place is not of type. Return NIL.

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

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

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

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

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

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

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

(not type)  → Complement of type.

(type* a)  → Type specifier for intersection of types.

(or type b)  → Type specifier for union of types.

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

•  → As a type argument (cf. Figure 2): no restriction.
10 CLOS

10.1 Classes

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

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

(defclass foo (superclass (standard-object))
  slot
  {reader reader}*
  {writer writer}*
  {self writer}*
  {accessor accessor}*
  {allocation instance}:
  {class (standard-class)}
  {initarg :instance-name}
  {initialform form}
  {type type}
  {documentation slot-doc}
  {default-initargs (name value)∗}
  {documentation class-doc}
  {metaclass name (standard-class)}
⇒ Define or modify class foo as a subclass of superclasses. Transform existing instances, if any, by \texttt{make-instance-obsolete}. In a new instance \texttt{i} of \texttt{foo}, a slot’s value defaults to \texttt{form} unless set via \texttt{:instance-name}; it is readable via (\texttt{reader i}) or (\texttt{accessor i}), and writable via (\texttt{writer value i}) or (\texttt{self accessor i} value). slots with \texttt{allocation} \texttt{class} are shared by all instances of class \texttt{foo}.

(find-class symbol \texttt{errorp} (environment)) ⇒ Return class named \texttt{symbol}, \texttt{setf}able.

(make-instance class \{\texttt{:instance-name} other-keyarg∗\}) ⇒ Make new instance of class.

(reinitialize-instance instance \{\texttt{:instance-name} other-keyarg∗\}) ⇒ Change local slots of instance according to instance by means of \texttt{shared-initialize}.

(slot-value foo slot) ⇒ Return value of slot in \texttt{foo}, \texttt{setf}able.

(slot-makunbound instance slot) ⇒ Make slot in \texttt{instance} unbound.

\{with-slots \{slot \texttt{var slot}∗\} \}
\{with-accessors \{\texttt{var accessor}∗\} \}
⇒ Return values of \texttt{forms} after evaluating them in a lexical environment with slots of instance visible as \texttt{setf}able slots or \texttt{var}/with-accessors of instance visible as \texttt{setf}able vars.

(class-name class)

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

(class-of foo) ⇒ Class \texttt{foo} is a direct instance of.

(change-class instance new-class \{\texttt{:instance-name} other-keyarg∗\}) ⇒ Change class of \texttt{instance} to \texttt{new-class}. Retain the status of any slots that are common between \texttt{instance}’s original class and \texttt{new-class}. Initialize any newly added slots with the values of the corresponding \texttt{initargs} if any, or with the values of their \texttt{initialform} forms if not.

(make-instances-obsolete)
⇒ Update all existing instances of \texttt{class} using \texttt{update-instance-for-redefined-class}.

\{initialize-instance instance \}
\{update-instance-for-different-class previous current \}
⇒ Set slots on behalf of \texttt{make-instance}/\texttt{change-class} by means of \texttt{shared-initialize}.

(m-restart-bind ((restart \texttt{KILL}) restart-function)
  :interactive-function arg-function
  :report-function report-function
  :test-function test-function)
⇒ Return values of \texttt{forms} evaluated with dynamically established \texttt{restarts} whose \texttt{restart-functions} should perform a non-local transfer of control. A restart is visible under condition if \texttt{(test-function condition)} 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\textsuperscript{*})}, 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}.

(invoke-restart restart arg\textsuperscript{*})

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

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

\{restart-name restart\}
⇒ Name of restart.

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

(with-condition-restarts condition \texttt{restarts form\textsuperscript{*}})
⇒ Evaluate \texttt{forms} with \texttt{restarts} dynamically associated with \texttt{condition}. Return values of \texttt{forms}.

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

\{cell-error-name \}
⇒ Name of cell which caused \texttt{condition}.

\{unbound-slot-instance \}
⇒ Instance with unbound slot which caused \texttt{condition}.

\{print-not-readable-object \}
⇒ The object not readable printable under \texttt{condition}.

\{package-error-package \}
\{file-error-pathname \}
\{stream-error-stream \}
⇒ Package, path, or stream, respectively, which caused the \texttt{condition} of indicated type.

\{type-error-datum \}
\{type-error-expected-type \}
⇒ Object which caused \texttt{condition} of type \texttt{type-error}, or its expected type, respectively.

\{simple-condition-format-control \}
\{simple-condition-format-arguments \}
⇒ Return \texttt{format} control or list of \texttt{format} arguments, respectively, of \texttt{condition}.

\{break-on-signals \}
⇒ Condition type debugger is to be invoked on.
(\(\texttt{make-condition}\) \texttt{condition-type \{\texttt{t}\} \texttt{condition-name}})
\> Return new instance of \texttt{condition-type}.

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

(\texttt{error continue-control} \texttt{condition continue-arg} \texttt{control arg}^\texttt{T})
\> Unless handled, \texttt{signal} as \texttt{error} or a new instance of \texttt{condition-type} or, with \texttt{format} control and args (see page 36), \texttt{simple-error}. In the debugger, use \texttt{format} arguments \texttt{continue-control} and \texttt{continue-arg} to tag the continue option. Return \texttt{NIL}.

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

(\texttt{invoke-debugger condition})
\> Invoke debugger with \texttt{condition}.

(\texttt{assert test \{\texttt{place}\}} \texttt{condition continue-arg} \texttt{control arg}^\texttt{T})
\> If test, which may depend on \texttt{places}, returns \texttt{NIL}, \texttt{signal} as \texttt{correctable error} \texttt{condition} or a new instance of \texttt{condition-type} or, with \texttt{format} control and args (see page 36), \texttt{error}. When using the debugger's continue option, \texttt{places} can be altered before re-evaluation of \texttt{test}. Return \texttt{NIL}.

(\texttt{handler-case foo} \texttt{(type \{\texttt{var}\}) \{\texttt{decl}\}} \texttt{condition-form})
\> If, on evaluation of \texttt{foo}, a condition of \texttt{type} is signalled, evaluate matching \texttt{condition-forms} with \texttt{var} bound to the condition, and return their values. Without a condition, \texttt{bind \texttt{ord-\&} to values of \texttt{foo}} and return values of \texttt{forms} or, without a \texttt{no-error} clause, return values of \texttt{foo}. See page 17 for \texttt{ord-\&}.

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

(\texttt{with-simple-restart} \texttt{(restart \{\texttt{nil}\} control arg}) \texttt{form})
\> Return values of \texttt{forms} unless \texttt{restart} is called during their evaluation. In this case, describe \texttt{restart} using \texttt{format} control and args (see page 36) and return \texttt{NIL} and \texttt{T}.

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

(\texttt{update-instance-for-redefined-class} \texttt{new-instance added-slots discarded-slots discarded-slots-property-list})
\> On behalf of \texttt{make-instances-obsolete} and by means of \texttt{\#shared-initialize}, set any \texttt{initarg} slots to their corresponding values; set any remaining \texttt{added-slots} to the values of their \texttt{initform} forms. Not to be called by user.

(\texttt{allocate-instance class \{\texttt{t}\} \{\texttt{initarg-slot-value}\}}
\> Return uninitialized instance of class. Called by \texttt{\#make-instance}.

(\texttt{slot-missing class instance slot \{\texttt{slot-boundp}\} \texttt{slot-makunbound \{\texttt{value}\}}
\> Called on attempted access to non-existing or unbound slot. Default methods signal \texttt{error/unbound-slot}, respectively. Not to be called by user.

10.2 Generic Functions

(\texttt{next-method-p})
\> \texttt{T} if enclosing method has a next method.

(\texttt{defgeneric foo \{\texttt{setf foo}\}} \texttt{(required-var \&optional \{\texttt{var} \texttt{[var]\}} \texttt{[krest \texttt{var}] \texttt{[kkey \texttt{[key var]}]} \texttt{[allow-other-keys]}\}}
\> Define or modify generic function \texttt{foo}. Remove any methods previously defined by \texttt{defgeneric}. \texttt{gf-class} and the lambda parameters \texttt{required-var} and \texttt{var} must be compatible with existing methods. \texttt{defmethod-args resemble those of \#defmethod}. For \texttt{c-type} see section 10.3.

(\texttt{ensure-generic function foo \{\texttt{setf foo}\}}
\> Define or modify generic function \texttt{foo}. \texttt{gf-class} and \texttt{lambda-list} must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to \texttt{method-class} do not propagate to existing methods. For \texttt{c-type} see section 10.3.
Define new method for generic function `foo`. spec-vars specialize to either being of class or being `eq` bar, respectively. On invocation, vars and spec-vars of the new method act like parameters of a function with body `form`. forms are enclosed in an implicit `block foo`. Applicable qualifiers depend on the `method-combination` type; see section 10.3.

Define new method for generic function `foo`. spec-vars specialize to either being of class or being `eq` bar, respectively. On invocation, vars and spec-vars of the new method act like parameters of a function with body `form`. forms are enclosed in an implicit `block foo`. Applicable qualifiers depend on the `method-combination` type; see section 10.3.

Define new method for generic function `foo`. spec-vars specialize to either being of class or being `eq` bar, respectively. On invocation, vars and spec-vars of the new method act like parameters of a function with body `form`. forms are enclosed in an implicit `block foo`. Applicable qualifiers depend on the `method-combination` type; see section 10.3.