# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Numbers</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>Numeric Functions</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Logic Functions</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>Integer Functions</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>Implementation-Dependent</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Characters</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Strings</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Conses</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Lists</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Association Lists</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Trees</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Sets</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Arrays</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Array Functions</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Vector Functions</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Sequences</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Sequence Predicates</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Sequence Functions</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Hash Tables</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Structures</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Control Structure</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>Variables</td>
<td></td>
</tr>
<tr>
<td>9.3</td>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>Macros</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CLOS</td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>Classes</td>
<td></td>
</tr>
<tr>
<td>10.2</td>
<td>Generic Functions</td>
<td></td>
</tr>
<tr>
<td>10.3</td>
<td>Method Combination Types</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Conditions and Errors</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Types and Classes</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Input/Output</td>
<td></td>
</tr>
<tr>
<td>13.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>13.2</td>
<td>Reader</td>
<td></td>
</tr>
<tr>
<td>13.3</td>
<td>Character Syntax</td>
<td></td>
</tr>
<tr>
<td>13.4</td>
<td>Printer</td>
<td></td>
</tr>
<tr>
<td>13.5</td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>13.6</td>
<td>Streams</td>
<td></td>
</tr>
<tr>
<td>13.7</td>
<td>Pathnames and Files</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Packages and Symbols</td>
<td></td>
</tr>
<tr>
<td>14.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>14.2</td>
<td>Packages</td>
<td></td>
</tr>
<tr>
<td>14.3</td>
<td>Symbols</td>
<td></td>
</tr>
<tr>
<td>14.4</td>
<td>Standard Packages</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Compiler</td>
<td></td>
</tr>
<tr>
<td>15.1</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>15.2</td>
<td>Compilation</td>
<td></td>
</tr>
<tr>
<td>15.3</td>
<td>REPL and Debugging</td>
<td></td>
</tr>
<tr>
<td>15.4</td>
<td>Declarations</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>External Environment</td>
<td></td>
</tr>
</tbody>
</table>

## Typographic Conventions

*name*; *name*; *name*; *name*; *name*; *name*; *name*; *name*; *name*; *name*

- Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.

*them*  
- Placeholder for actual code.

*me*  
- Literal text.

{*foo*}  
- Either one *foo* or nothing; defaults to *bar*.

*foo*; {*foo*}  
- Zero or more *foos*.

*foo*; {*foo*}  
- One or more *foos*.

*foos*  
- English plural denotes a list argument.

{*foo|bar|baz*}; {*foo|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 20.

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

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

1.1 Predicates

\( (= \text{number}^+ \) \) \( \iff \) If all numbers, or none, respectively, are equal in value.

\( (=/= \text{number}^+ \) \) \( \iff \) If \( a = 0 \), or \( a > 0 \), respectively.

\( (= \text{number}^- \) \) \( \iff \) If \( a - \text{number} > 0 \) is even or odd, respectively.

\( (= \text{number}^- \) \) \( \iff \) if int is even or odd, respectively.

\( (= \text{number}^- \) \) \( \iff \) or \( b \).

\( (= \text{number}^- \) \) \( \iff \) Return \( a - \text{number} > 0 \), respectively. Without any bs, remove \( a \) or \( b \), respectively.

\( (= \text{number}^- \) \) \( \iff \) Return \( a + 1 \) or \( a - 1 \), respectively.

\( (= \text{number}^- \) \) \( \iff \) Increment or decrement the value of place by delta. Return new value.

\( (= \text{number}^- \) \) \( \iff \) Return \( a \) or \( b \), respectively.

\( (= \text{number}^- \) \) \( \iff \) Return \( \log a \), or, without \( b \), \( \ln a \).

\( (= \text{number}^- \) \) \( \iff \) or \( \sqrt n \) in complex numbers/natural numbers.

\( (= \text{number}^- \) \) \( \iff \) Least common multiple or greatest common denominator, respectively. of integers. \( (gcd \text{integer} \) \( \iff \) \( \pi \) \( \iff \) long-float approximation of \( \pi \), Ludolph's number.

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

\( (\tan a \) \( \iff \) arctan \( a \) + radians.

\( (\sin a \) \( \iff \) sin \( a \) or \( \cos a \), respectively. 

\( (\tan a \) \( \iff \) 

\( (\sinh a \) \( \iff \) 

\( (\cosh a \) \( \iff \) 

\( (\tanh a \) \( \iff \) 

\( (\text{atan a} \) \( \iff \) 

\( (\text{atan a} \) \( \iff \)
Number of magnitude 1 representing sign or phase of rational. Assume complete/limited accuracy for real.

Float real [prototype float] ▶ Convert real into float with type of prototype.

1.3 Logic Functions

Negative integers are used in two’s complement representation.

Boole operation int-a int-b ▶ Return value of bitwise logical operation. operations are:

Boole-2 ▶ int-2.
Boole-1 ▶ int-1.
Boole-e2 ▶ ~int-2.
Boole-set ▶ All bits set.
Boole-clr ▶ All bits zero.
\begin{itemize}
\item \texttt{bits in} \texttt{int-a} \texttt{=} \texttt{int-b} \\
\item \texttt{=\texttt{-int-a \& \texttt{-int-b)}} \\
\item \texttt{-int-a \texttt{\&\texttt{-int-b}}} \\
\item \texttt{} \\
\item \texttt{} \\
\end{itemize}

\section*{1.4 Integer Functions}

\subsection*{integer-length integer} \textbf{Return} \texttt{size} \textbf{of} \texttt{integer}.

\subsection*{lbd-test byte-type integer} \textbf{Return} \texttt{true} \textbf{if} \texttt{any} \texttt{bits} \textbf{represent} \texttt{byte-type} \textbf{integer}.

\subsection*{ash integer} \textbf{Return} \texttt{integer} \textbf{arithmetic} \textbf{shift} \texttt{left} \texttt{bit} \texttt{adding} \texttt{zeros} \texttt{at} \texttt{right}, \texttt{or}, \texttt{for} \texttt{count < 0}, \texttt{shifted} \texttt{right} \texttt{discarding} \texttt{bits}.

\subsection*{byte-type integer} \textbf{Return} \texttt{integer} \textbf{arithmetic} \textbf{shift} \texttt{left} \texttt{bit} \texttt{adding} \texttt{zeros} \texttt{at} \texttt{right}, \texttt{or}, \texttt{for} \texttt{count < 0}, \texttt{shifted} \texttt{right} \texttt{discarding} \texttt{bits}.

\subsection*{byte-size integer} \textbf{Return} \texttt{byte} \textbf{size} \textbf{of} \texttt{integer} \textbf{bit} \texttt{size} \texttt{starting} \texttt{at} \texttt{weight} \texttt{position}.

\subsection*{byte-size byte-type} \textbf{Return} \texttt{byte} \textbf{size} \textbf{of} \texttt{integer} \textbf{bit} \texttt{size} \texttt{starting} \texttt{at} \texttt{weight} \texttt{position}.

\subsection*{byte-size \textit{position}} \textbf{Return} \texttt{byte} \textbf{size} \textbf{of} \texttt{integer} \textbf{bit} \texttt{size} \texttt{starting} \texttt{at} \texttt{weight} \texttt{position}.

\subsection*{byte-size \textit{position}} \textbf{Return} \texttt{byte} \textbf{size} \textbf{of} \texttt{integer} \textbf{bit} \texttt{size} \texttt{starting} \texttt{at} \texttt{weight} \texttt{position}.
1.5 Implementation-Dependent

- short-float
- single-float
- double-float
- long-float
  ▶ Smallest possible number making a difference when added or subtracted, respectively.

- least-negative
- least-negative-normalized
- least-positive
- least-positive-normalized
  ▶ Available numbers closest to 0 or +0, respectively.

- most-negative
- most-positive
  ▶ Available numbers closest to −∞ or +∞, respectively.

\( \text{c} \)

1.5 Implementation-Dependent

- short-float
- single-float
- double-float
- long-float
  ▶ Smallest possible number making a difference when added or subtracted, respectively.

- least-negative
- least-negative-normalized
- least-positive
- least-positive-normalized
  ▶ Available numbers closest to 0 or +0, respectively.

- most-negative
- most-positive
  ▶ Available numbers closest to −∞ or +∞, respectively.

\( \text{c} \)

2 Characters

The standard-char type comprises a-z, A-Z, 0-9, Newline, Space, and !?"'\&':;\*+/\-\-\<\>\@\#\%\&(){}.

\( \text{c} \)

2 Characters

The standard-char type comprises a-z, A-Z, 0-9, Newline, Space, and !?"'\&':;\*+/\-\-\<\>\@\#\%\&(){}.

\( \text{c} \)

16 External Environment

\( \text{c} \)

16 External Environment

\( \text{c} \)
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` ➤ NIL if `foo` is of indicated type.

```lisp
(defparameter string-not-lessp character+ string-lessp character+) ➤ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
```

```lisp
(char-upcase character) (char-downcase character) ➤ Return corresponding uppercase/lowercase character, respectively.
```

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` ➤ NIL if `foo` is of indicated type.

```lisp
(defparameter string-not-lessp character+ string-lessp character+) ➤ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
```

```lisp
(char-upcase character) (char-downcase character) ➤ Return corresponding uppercase/lowercase character, respectively.
```

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` ➤ NIL if `foo` is of indicated type.

```lisp
(defparameter string-not-lessp character+ string-lessp character+) ➤ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
```

```lisp
(char-upcase character) (char-downcase character) ➤ Return corresponding uppercase/lowercase character, respectively.
```

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` ➤ NIL if `foo` is of indicated type.

```lisp
(defparameter string-not-lessp character+ string-lessp character+) ➤ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
```

```lisp
(char-upcase character) (char-downcase character) ➤ Return corresponding uppercase/lowercase character, respectively.
```
4 Conses

4.1 Predicates

(,%cons foo)  ▶ Return T if foo is of indicated type.

(,%listp foo)  ▶ Return T if list/foo is NIL.

(,%endp list)  ▶ Return T if list/foo is a cons.

(,%null foo)  ▶ Return T if foo is not a cons.

(,%tailp foo list)  ▶ Return T if foo is a tail of list.

(,%member foo list)  ▶ Return tail of list starting with its first element matching foo. Return NIL if there is no such element.

(,%member-if member-if-not)  ▶ Return tail of list starting with its first element satisfying test. Return NIL if there is no such element.

(,%subsetp list-a list-b)  ▶ Return T if list-a is a subset of list-b.

4.2 Lists

(,%cons foo bar)  ▶ Return new cons (foo . bar).

(,%list foo*)  ▶ Return list of foos.

(,%list+ foo+)  ▶ Return list of foos with last foo becoming cdr of last cons. Return foo if only one foo given.

(,%make-list num [initial-element foo])  ▶ New list with num elements set to foo.

(,%length list)  ▶ Length of list; NIL for circular list.

(,%car list)  ▶ Car of list or NIL if list is NIL. setfable.

(,%cdr list)  ▶ Cdr of list or NIL if list is NIL. setfable.

(,%rest list)  ▶ Return tail of list after calling cdr n times.

(,%nth n list)  ▶ Zero-indexed n-th element of list. setfable.

(,%cadr list)  ▶ With X being one to four as and ds representing ,cars and ,cdrs, e.g. (,cadr bar) is equivalent to (,car (,cdr bar)). setfable.

(,%last list [num])  ▶ Return list of last num conses of list.
14.4 Standard Packages

common-lisp

➢ exports the defined names of Common Lisp except for those in the keyword package.

common-lisp-user

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

keyword

➢ contains symbols which are defined to be of type keyword.

15 Compiler

15.1 Predicates

(defun special-operator-p foo)
  ➢ T if foo is a special operator.

(defun compiled-function-p foo)
  ➢ T if foo is of type compiled-function.

15.2 Compilation

(defun compile \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
  \end{itemize}
\end{quote}
  ➢ 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.

(defun compile-file \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{output-file out-path}
    \item \texttt{print \{compile-verbose\} \{compile-print\} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ Write compiled contents of file to out-path. Return true output path or NIL T in case of warnings or errors, T in case of warnings or errors excluding style-warnings.

(defun compile-file-pathname \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{output-file out-path} \{other-keyargs\}
  \end{itemize}
\end{quote}
  ➢ Pathname compile-file writes to if invoked with the same arguments.

(defun load \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ Load source file or compiled file into Lisp environment. Return T if successful.

(defun *compile-file* \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ Input file used by \texttt{compile-file} by \texttt{load}.

(defun *compile* \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ defaults used by \texttt{compile-file} by \texttt{load}.

4.3 Association Lists

(defun pairlis \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ Prepend to \texttt{alist} an association list made from \texttt{keys} and \texttt{values}.

(defun acons \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ Return \texttt{alist} with a \texttt{(key, value)} pair added.

(defun assoc \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ First cons whose car, or cdr, respectively, satisfies \texttt{test}.

(defun copy-alist \begin{quote}
  \begin{itemize}
    \item \texttt{NIL definition name \{setf name\} \{definition\}}
    \item \texttt{load-path \{load-verbose\} \{load-print\} \{if-does-not-exist \{load-exists\} \} \{external-format file-format\}}
  \end{itemize}
\end{quote}
  ➢ Return \texttt{copy} of \texttt{alist}.
4.4 Trees

\[ \textlangle \text{tree-equal} \text{ foo bar} \rangle \]

\[ \textlangle \text{tree-not} \text{ foo bar} \rangle \]

\[ \text{// Return } \top \text{ if } \text{foo and bar have same shape and leaves satisfying } \text{Test.} \]

\[ \text{\{\text{test function } \text{xor}\} \{\text{test-not function } \text{xor}\}} \]

\[ \text{// Make copy of tree with each subtree or leaf matching } \text{old} \text{ replaced by } \text{new}. \]

\[ \text{\{\text{test function } \text{and}\} \{\text{test-not function } \text{and}\}} \]

\[ \text{// Make copy of tree with each subtree or leaf satisfying } \text{test} \text{ replaced by } \text{new}. \]

\[ \text{\{\text{test function } \text{or}\} \{\text{test-not function } \text{or}\}} \]

\[ \text{// Make copy of tree with each subtree or leaf matching } \text{a key in } \text{association-list} \text{ replaced by that key's value.} \]

\[ \text{// Copy of tree with same shape and leaves.} \]

4.5 Sets

\[ \text{// (\text{test function } \text{xor})} \]

\[ \text{// (\text{test-not function } \text{xor})} \]

\[ \text{// Return } a \cap b, a \setminus b, a \cup b, \text{ or } a \triangle b, \text{ respectively, of lists } a \text{ and } b. \]

5 Arrays

5.1 Predicates

\[ \text{// (\text{array } \text{foo})} \]

\[ \text{// (\text{vectorp } \text{foo})} \]

\[ \text{// (\text{simple-vector-p } \text{foo})} \]

\[ \text{// (\text{bit-vector-p } \text{foo})} \]

\[ \text{// (\text{adjustable-array-p } \text{array})} \]

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

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

\[ \text{// Return } \top \text{ if subscripts are in array's bounds.} \]

5.2 Array Functions

\[ \text{// (\text{make-array } \text{dimension-sizes } \{\text{:adjustable boolean}\}} \]

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

\[ \text{// (\text{fill-pointer } \{\text{null boolean}\}} \]

\[ \text{// (\text{initial-contents tree-or-array}} \]

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

\[ \text{// Return fresh, or readjust, respectively, vector or array.} \]

\[ \text{// Return array element pointed to by subscripts. } \text{settable.} \]

\[ \text{// Return } n \text{th element of } \text{array in row-major order. } \text{settable.} \]

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).

\[ \text{// (\text{make-symbol } \text{name})} \]

\[ \text{// Return fresh, uninterned symbol } \text{name}. \]

\[ \text{// (\text{gensym } \text{\#n})} \]

\[ \text{// Return fresh, uninterned symbol } \#\text{an with } n \text{ from } \text{*gensym-counters*}. \text{ Increment } \text{*gensym-counters*}. \]

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

\[ \text{// Intern fresh symbol in package. Deprecated.} \]

\[ \text{// (\text{copy-symbol } \text{symbol } \{\text{properties}\})} \]

\[ \text{// Return uninterned copy of symbol. If } \text{props is } \top \text{, give copy the same value, function and property list.} \]

\[ \text{// (\text{symbol-name } \text{symbol})} \]

\[ \text{// Name or package, respectively, of symbol.} \]

\[ \text{// (\text{symbol-package symbol})} \]

\[ \text{// Property list, value, or function, respectively, of symbol. } \text{settable.} \]
14 Packages and Symbols

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

14.1 Predicates

(symbolp foo) ⇒ T if foo is of indicated type.

(packagep foo) ⇒ Exported symbol of package.

(keywordp foo) ⇒ Possibly unexported symbol of package.

-packagep foo ⇒ 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 ⇒ Create package foo.


-in-package foo ⇒ Make package foo current.

-use-package other-packages [package-packages] ⇒ 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* ⇒ 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.

-find-symbol foo [package packages] ⇒ Intern or find, respectively, symbol foo in package. Second return value is one of internal, external, or inherited (or NIL if intern has created a fresh symbol).

-unintern symbol [package packages] ⇒ Remove symbol from package, return T on success.

14.2 Packages

:bar | keyword :bar ⇒ Keyword, evaluates to :bar.

package :symbol ⇒ Exported symbol of package.

package :symbol ⇒ Possibly unexported symbol of package.

nicknames (nick*) ⇒ (documentation string)

( intern interned-symbol*) ⇒ (use used-package*)

(import-from pkg imported-symbol*) ⇒ (shadowing-import-from pkg shd-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 ⇒ Create package foo.


-in-package foo ⇒ Make package foo current.

-use-package other-packages [package-packages] ⇒ 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* ⇒ 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.

-find-symbol foo [package packages] ⇒ Intern or find, respectively, symbol foo in package. Second return value is one of internal, external, or inherited (or NIL if intern has created a fresh symbol).

-unintern symbol [package packages] ⇒ Remove symbol from package, return T on success.
6 Sequences

6.1 Sequence Predicates

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

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

\[mismatch sequence-a sequence-b\] \[\{:from-end\} \{test function\} \{:test-not function\} \{start1 start2\} \{end1 end2\}\] → Return position in sequence-a where sequence-a and sequence-b begin to mismatch. Return NIL if they match entirely.

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).

\{from-end \} \{test function\} \{start\} \{end\} \{key function\} → Return number of elements in sequence which match foo.

\{count \} \{count-if \} \{count-if-not \} test sequence \{from-end \} \{start \} \{end\} \{key function\} → Return number of elements in sequence which satisfy test.

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

\{subseq sequence start \{end\}\} → Return subsequence of sequence between start and end. setfable.

\{sort \} \{stable-sort \} sequence test \{key function\} → Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

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

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

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

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

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

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

((stream-external-format stream)
  ▷ External file format designator.
)

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

13.7 Pathnames and Files

(make-pathname
  (host (host NIL unspecific))
  :device (device NIL unspecific)
  (directory (directory NIL unspecific)
    (directory NIL unspecific)
    (directory NIL unspecific)
  )
  :name (file-name NIL unspecific)
  :type (file-type NIL unspecific)
  :version (version NIL unspecific)
  :defaults path (from (relative path) (absolute path) (directory path)
    (case :local common)
  )
  ▷ Construct a logical pathname if there is a logical pathname translation for host, otherwise construct a physical pathname. For :case :local, leave case of components unchanged; convert all-uppercase components into local customary case; do the opposite with all-lowercase components.
)

(pathname-host
  pathname-device
  (pathname-directory
    pathname-name
    pathname-type)
  path-or-stream (case :local :common)
  ▷ Return pathname component.
)

(find sequence
  (find position foo sequence)
  :test function
  :test-not function
  :start start
  :send end
  :key function
  ▷ Return first element in sequence which satisfies test, or its position relative to the begin of sequence, respectively.
)

(find-if
  (find-if-not
    (find-if position foo test sequence)
    :test function
    :test-not function
    :start start
    :send end
    :key function
    ▷ Return first element in sequence which matches foo, or its position relative to the begin of sequence, respectively.
)

(search sequence-a sequence-b
  start1
  end1
  start2
  end2
  :test function
  :test-not function
  :key function
  ▷ Search sequence-b for a subsequence matching sequence-a. Return position in sequence-b, or NIL.
)

(remove foo sequence
  :test function
  :test-not function
  :start start
  :send end
  :key function
  :count count
  ▷ Make copy of sequence without elements matching foo.
)

(remove-if
  (remove-if-not
    (remove-if test sequence)
    :test function
    :test-not function
    :start start
    :send end
    :key function
    :count count
    ▷ Make copy of sequence with all (or count) elements satisfying test removed.
)

(remove-duplicates sequence
  (remove-duplicates sequence)
  :test function
  :test-not function
  :start start
  :send end
  :key function
  :count count
  ▷ Make copy of sequence without duplicates.
)

(substitute new old sequence
  :test function
  :test-not function
  :start start
  :send end
  :key function
  :count count
  ▷ Make copy of sequence with all (or count) olds replaced by new.
)

(substitute-if
  (substitute-if-not
    (substitute-if new test sequence)
    :test function
    :test-not function
    :start start
    :send end
    :key function
    :count count
    ▷ Make copy of sequence with all (or count) elements satisfying test replaced by new.
)

(replace sequence-a sequence-b
  start1
  start2
  end1
  end2
  ▷ Replace elements of sequence-a with elements of sequence-b.
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.

(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.

(copies seq sequence)  
▷ Copy of sequence with shared elements.

39
8 Structures

(definestructure foo)

|conc-name [slot-prefix conc] | constructor [maker make-foo (ord-\lambda)] | copier [copy-copy] | (include struct slot \{slot init :type sl-type \{read-only \tilde{\delta}\}\}) |

|\{conc-name [slot-prefix conc]\} | constructor [maker make-foo (ord-\lambda)] | copier [copy-copy] | (include struct slot \{slot init :type sl-type \{read-only \tilde{\delta}\}\}) |

\{foo\}

▷ Define structure foo together with functions MAKE-foo, COPY-foo and foo-P and setable 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-\lambda (see page 17) is given, by (maker arg \{* \key value\} ). In the latter case, args and \texttt{keys} correspond to the positional and keyword parameters defined in ord-\lambda whose vars in turn correspond to slots. (print-object)/print-function generate a \texttt{print-object} method for an instance bar of foo calling (o-printer bar stream) or (f-printer bar stream print-level), respectively. If \texttt{type} without :named is given, no foo-P is created.

▷ Return copy of structure with shared slot values.

9 Control Structure

9.1 Predicates

(\texttt{eq \texttt{foo bar}}) ▷ T if \texttt{foo} and \texttt{bar} are identical.

(\texttt{eq \texttt{foo bar}}) ▷ T if \texttt{foo} and \texttt{bar} are identical, or the same character, or numbers of the same type and value.

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

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

(\texttt{not \texttt{foo}}) ▷ T if \texttt{foo} is \texttt{NIL}; \texttt{NIL} otherwise.

(\texttt{boundp symbol}) ▷ T if \texttt{symbol} is a special variable.

(\texttt{constantp \texttt{foo \{environment\}}}) ▷ T if \texttt{foo} is a constant form.
9.2 Variables

- `(defconstant symbol form [doc])` 
  Assign value of form to global constant/dynamic variable symbol.
- `(defparameter symbol form [doc])` 
  Assign value of form to dynamic variable symbol.
- `(defvar symbol form [doc])` 
  Unless bound already, assign value of form to dynamic variable symbol.
- `(setf [psetf] symbol form)` 
  Set places to primary values of forms. Return values of last form/NIL; work sequentially/in parallel, respectively.
- `(setq [psetq] symbol form)` 
  Set symbols to primary values of forms. Return value of last form/NIL; work sequentially/in parallel, respectively.
- `(set symbol foo)` 
  Set symbol’s value cell to foo. Deprecated.
- `(multiple-value-setq vars form)` 
  Set elements of vars to the values of form. Return form’s primary value.
- `(shift place+ foo)` 
  Store value of foo in rightmost place shifting values of places left, returning first place.
- `(rotatef place*)` 
  Rotate values of places left, old first becoming new last place’s value. Return NIL.
- `(makunbound foo)` 
  Delete special variable foo if any.
- `(get key [default])` 
  Return key and value of first entry from property list stored in symbol/in place, respectively, or default if there is no key. Setfable.
- `(getf place key [default])` 
  First entry key from property list stored in symbol/in place.
- `(get-properties property-list)` 
  Return key and value of first entry from property-list matching a key in key, and tail of property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.
- `(remprop symbol key)` 
  Remove first entry key from property list stored in symbol/in place, respectively. Return T if key was there, or NIL otherwise.
- `(progv symbols values form)` 
  Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.
- `(let [let*] ([name [value default]]) (declare dec*)* form)` 
  Evaluate forms with names lexically bound (in parallel or sequentially, respectively) to values. Return values of forms.
- `(multiple-value-bind (var*) values-form (declare dec*)* body-form)` 
  Evaluate body-forms with vars lexically bound to the return values of values-form. Return values of body-forms.
9.3 Functions

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

\[
\text{(var}^* \quad \text{[optional \quad} \text{var} \quad \text{[var \quad} \text{init} \quad \text{supplied-p]} \quad \text{]} \quad \text{[rest \quad var]}
\]

and \&environment clause.

\[
\text{(mdefun \quad foo \quad (ord-λ*)} \quad \text{[declare \quad decl*] \quad \text{[form*]}}
\]

\[
\text{mlambda \quad (ord-λ*)} \quad \text{form*} \quad \text{[doc]} \quad \text{\&environment}
\]

\[
\text{mdefun forms are enclosed in an implicit \&block named foo.}
\]

\[
\text{mlet \quad \text{labels} \quad \{\text{(foo \quad (ord-λ*)}} \quad \text{(setf \quad foo) \quad (new-value \quad ord-λ*)}} \quad \text{(declare \quad local-decl*) \quad \text{[doc]} \quad \text{form*}} \quad \text{[local-form*]}} \quad \text{(declare \quad decl*)} \quad \text{form*} \quad \text{[doc]}
\]

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

\[
\text{(function \quad \text{foo} \quad \text{[\&lambda \quad \text{form*}]]}}
\]

\[
\text{Return lexically innermost function named foo or a lexical closure of the \&lambda \quad \text{expression.}}
\]

\[
\text{(apply \quad \text{function}} \quad \text{[setf \quad function]} \quad \text{arg}^* \quad \text{args}} \quad \text{Values of function called with args and the list elements of args. setfable if function is one of \&ref, \&bit, and \&rbit.}
\]

\[
\text{(funcall \quad \text{function}} \quad \text{arg}^* \quad \text{Values of function called with args.}}
\]

\[
\text{(multiple-value-call \quad \text{function \quad form*}]} \quad \text{Call function with all the values of each form as its arguments. Return values returned by function.}
\]

\[
\text{(values-list \quad list \quad \text{Elements of list.}}
\]

\[
\text{\text{return \quad \text{as multiple values the primary values of the foos. setfable.}}}
\]

\[
\text{(multiple-value-list \quad form \quad \text{List of the values of form.}}
\]

\[
\text{(nth-value \quad n \quad form \quad \text{Zero-indexed nth return value of form.}}
\]

\[
\text{(complement \quad function \quad \text{Return new function with same arguments and same side effects as function, but with complementary truth value.}}
\]

\[
\text{(constantly \quad \text{foo} \quad \text{Function of any number of arguments returning foo.}}
\]

\[
\text{(identity \quad \text{foo} \quad \text{Return foo.}}
\]
\[\text{(function-lambda-expression function)}\]
\[\text{\(\triangleright\) If available, return lambda expression of function, NIL if function was defined in an environment without bindings, and name of function.}\]

\[\text{//definition \{\text{foo \text{(setf foo)}\}}}\]
\[\text{\(\triangleright\) Definition of global function foo. settable.}\]

\[\text{(/makunbound foo)}\]
\[\text{\(\triangleright\) Remove global function or macro definition foo.}\]

\[\text{(/call-arguments-limit)}\]
\[\text{\(\triangleright\) Upper bound of the number of function arguments or lambda list parameters, respectively; \(\geq 50\).}\]

\[\text{(/multiple-values-limit)}\]
\[\text{\(\triangleright\) Upper bound of the number of values a multiple value can have; \(\geq 20\).}\]

\[\text{9.4 Macros}\]

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

or
\[
\begin{aligned}
&\{\text{&whole var} \ E \var \{\text{macro-}\lambda^*}\} \ E \{\text{&optional \var} \{\text{macro-}\lambda^*}\} \ E \\
&\{\text{&key \{\text{var} \{\text{macro-}\lambda^*}\} \ E}
\end{aligned}
\]

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

\[\text{(/define-macro)}\]
\[\text{\{\text{function} \{\text{macro-}\lambda^*}\} \ E \{\text{\text{macro-}\lambda^*\\}\} \ E \{\text{macro-}\lambda^\text{*}}\]
#### 13.4 Printer

```lisp
(print) → No output.
(print *stream*) → Print on stream.
(fprn *stream*) → Prin1 on stream.
(fprn *stream* 'extended-output) → Prin1 on stream and write an extended output form.
```

- `fprn` and `print` form are used for printing streams with the `extended-output` form.
- `print` is used for outputting to the standard output stream.
- `fprn` is used for printing to a specified stream.

#### 13.5 Control Flow

```lisp
(if test then |else|) → Return values of `then` if `test` returns T; return values of `else` otherwise.

(when |unbreakable| test) → Return the values of the first `then*` whose `test` returns T; return NIL if all `test` returns NIL.
```

- `if` and `when` control the flow of execution based on conditions.
- `when` is used for executing blocks conditionally.
- `unbreakable` variant of `if` ensures the execution of blocks regardless of results.

---

**Common Lisp Quick Reference**

\[
\left\{ \begin{array}{l}
(m.n) \in \{([P]\{D\}E)^2] \in \mathbb{N}\} \\{([P]\{D\}E)^x \in \mathbb{N}\} \\
\end{array} \right.
\]

- `m.n` may represent non-integer values.

---

**Common Lisp Quick Reference**

\[
(\text{#c}(a \ b)) \quad \Rightarrow \quad (\text{complex } a \ b) \quad \text{the complex number } a + bi.
\]

- `#c` is used to represent complex numbers.

---

**Common Lisp Quick Reference**

\[
(\text{#f} \text{foo}) \quad \Rightarrow \quad (\text{function } \text{foo}); \text{the function named } \text{foo}.
\]

- `#f` is used to denote functions.

---

**Common Lisp Quick Reference**

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

- `#n` is used to denote vectors.

---

**Common Lisp Quick Reference**

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

- `#n*` is used to denote bit vectors.

---

**Common Lisp Quick Reference**

\[
\text{(defun} \text{macro *)}\quad\Rightarrow\quad(\text{declare} \text{doc}^*)\quad(\text{doc})
\]

- `defun` is used to define macros.

---

**Common Lisp Quick Reference**

\[
(\text{when} | |\text{unbreakable}| test) \quad \Rightarrow \quad \text{Evaluate } \text{foos} \text{ and return their values if } \text{test} \text{ returns } T \text{ or } \text{NIL}, \text{respectively. Return } \text{NIL} \text{ otherwise.}
\]

- `when` is used for evaluating blocks conditionally.
\[ (\text{case } \text{test } \{ \text{key} \} \text{ foo}^{0} + \{ \text{otherwise} \} \text{ bar}^{0} + \text{test}) \]

\text{Return the values of the first non-nil one of whose keys is eql test. Return values of \text{bars} if there is no matching key.}

\[ (\text{ecase } \{ \text{key} \} \text{ test } \{ \text{key} \} \text{ foo}^{0} + \text{test}) \]

\text{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.}

\[ (\text{and form}^{0}) \]

\text{Evaluate forms from left to right. Immediately return \text{NIL} if one form’s value is \text{NIL}. Return values of last form otherwise.}

\[ (\text{or form}^{0}) \]

\text{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 \text{NIL} if no form returns T.}

\[ (\text{progn form}^{0}) \]

\text{Evaluate forms sequentially. Return values of last form.}

\[ (\text{multiple-value-prog1 form-r form}^{0}) \]

\[ (\text{multiple-value-prog2 form-r form}^{0}) \]

\text{Evaluate forms in order. Return values/primary value, respectively, of form-r.}

\[ (\text{prog} \{ \text{name} \text{ value} \}^{0}) \]

\text{Evaluate \text{tagbody} like body with names lexically bound (in parallel or sequentially, respectively) to values. Return \text{NIL} or explicitly \text{return} values. Implicitly, the whole form is a \text{block} named \text{NIL}.}

\[ (\text{unwind-protect protected cleanup}^{0}) \]

\text{Evaluate protected and then, no matter how control leaves protected, cleanups. Return values of \text{protected}.}

\[ (\text{block name form}^{0}) \]

\text{Evaluate forms in a lexical environment, and return their values unless interrupted by \text{return-from}.}

\[ (\text{return-from} \text{foo result}^{0}) \]

\text{Have nearest enclosing \text{block} named \text{foo} named \text{NIL}, respectively, return with values of \text{result}.}

\[ (\text{tagbody} \{ \text{tag} \text{form}^{0} \}) \]

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

\[ (\text{go} \text{tag}) \]

\text{Within the innermost possible enclosing \text{tagbody}, jump to a tag \text{eq} tag.}

\[ (\text{catch} \text{tag form}^{0}) \]

\text{Evaluate forms and return their values unless interrupted by \text{throw}.}

\[ (\text{throw} \text{tag form}) \]

\text{Have the nearest dynamically enclosing \text{catch} with a tag \text{eq} tag return with the values of form.}

\[ (\text{sleep} \text{n}) \]

\text{Wait \text{n} seconds; return \text{NIL}.}
13 Input/Output

13.1 Predicates

\( \text{(streamp } \text{foo}) \)  \( \Rightarrow \text{T} \) if \( \text{foo} \) is of indicated type.

\( \text{(pathnamep } \text{foo}) \)  \( \Rightarrow \text{T} \) if \( \text{foo} \) is of indicated type.

\( \text{(readtablep } \text{foo}) \)  \( \Rightarrow \text{T} \) if \( \text{foo} \) is of indicated type.

\( \text{(input-stream-p } \text{stream}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(output-stream-p } \text{stream}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(interactive-stream-p } \text{stream}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(open-stream-p } \text{stream}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(pathname-match-p } \text{path } \text{wildcard}) \)  \( \Rightarrow \text{T} \) if \( \text{path} \) matches wildcard.

\( \text{(wild-pathname-p } \text{path } \text{predicate}) \)  \( \Rightarrow \text{T} \) if \( \text{predicate} \) indicates any component.

13.2 Reader

\( \text{(y-or-n-p } \text{arg}) \)  \( \Rightarrow \text{T} \) if \( \text{arg} \) is \text{t} or \text{nil} depending on their answer. See page 36, \text{format}, for control and args.

\( \text{(with-standard-io-syntax } \text{form}) \)  \( \Rightarrow \text{T} \) if \( \text{form} \) is a character, a list, a string, or a byte.

\( \text{(read } \text{stream} \text{ variance}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-preserving-whitespace } \text{stream} \text{ read-in}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-from-string } \text{var} \text{ stream} \text{ reculsive}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-deleted-list } \text{char} \text{ stream} \text{ read-in}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-char } \text{stream} \text{ read-in}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-char-no-whitespace } \text{stream} \text{ read-in}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(unread-char } \text{stream}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-byte } \text{stream} \text{ read-in}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.

\( \text{(read-line } \text{stream} \text{ read-in}) \)  \( \Rightarrow \text{T} \) if \( \text{stream} \) is for input, for output, interactive, or open, respectively.
12 Types and Classes

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

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

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

(type form) ▶ Declare values of form to be of type.

(coerce object type) ▶ Coerce object into type.

(mtypecase foo (type a-form^b)| (otherwise b-form^b)) ▶ Return values of the first a-form^b whose type is foo of. Return values of b-forms if no type matches.

(etypecase foo (type ^b)) ▶ Return values of the first ^b whose type is foo of. Signal non-convertible/convertible type-error if no type matches.

(of type foo) ▶ Type of foo.

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

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

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

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

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

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

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

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

(not type) ▶ Complement of type.

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

(or type*) ▶ Type specifier for union of types.

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

* ▶ As a type argument (cf. Figure 2): no restriction.

10 CLOS

10.1 Classes

(slot-finds-p foo bar) ▶ T if foo has a slot bar.
### (slot-boundp instance slot)
- if slot in instance is bound.

### (defclass foo (superclass* [standard-object])
- Define or modify class foo as a subclass of superclasses. Transform existing instances, if any, by `make-instances-obsolete`. In a new instance i of foo, a slot's value defaults to `form` unless set via `[:instance-name]`; it is readable via `(reader i)` or `(accessor i)`, and writable via `(writer value i)` or `(setf accessor i)`. Slots with :allocation `instance` are shared by all instances of class foo.

### (find-class symbol [error [environment]])
- Return class named `symbol`, `setfable`. If not, signal `*unbound-class*`.

### (make-instance [instance-name] other-keyarg*)
- Make new instance of class.

### (reinitialize-instance instance [:instance-name] other-keyarg*)
- Change local slots of instance according to initargs by means of `shared-initialize`.

### (slot-value foo slot)
- Return value of slot in `foo`, `setfable`.

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

### (with-slots ((slot [var slot])...) instance (declare decl*)
- Return values of forms after evaluating them in a lexical environment with slots of instance visible as `setfable` slots or `vars`/`with-accessors` of instance visible as `setfable` vars.

### (class-name class)
- Get/set name of class.

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

### (change-class instance new-class [:instance-name] other-keyarg*)
- Change class of `instance` to `new-class`. Retain the status of any slots that are common between `instance`s original class and `new-class`. Initialize any newly added slots with the values of the corresponding instargs if any, or with the values of their :initform forms if not.

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

### (initialize-instance instance)
- Set slots on behalf of `make-instance`/`change-class` by means of `shared-initialize`.

### (update-instance-for-redefined-class new-instance added-slots
discarded-slots discarded-slots-property-list
[:instance-name] other-keyarg*)
- On behalf of `make-instances-obsolete` and by means of `shared-initialize`, set any `initargs` slots to their corresponding values; set any remaining `added-slots` to the values of their :initform forms. Not to be called by user.

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

### (invoke-restart restart arg*)
- Call function associated with `restart` with arguments given or prompted for, respectively. If `restart` function returns, return its values.

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

### (restart-name restart)
- Name of `restart`.

### (abort)
- Call function associated with `abort` with arguments given or prompted for, respectively. If `restart` function returns, return its values.

### (muffle-warning continue store-value-value use-value-value)
- Control transfer to innermost applicable restart with same name (i.e. `abort`, `continue`) out of those either associated with `condition` or un-associated at all; or, without `condition`, out of all restarts. If no restart is found, signal `control-error` for `abort` and `muffle-warning`, or return `NIL` for the rest.

### (with-condition-restarts condition restarts form*)
- Evaluate forms with restarts dynamically associated with `condition`. Return values of forms.

### (arithmetic-error-operation condition)
- List of function or of its operands respectively, used in the operation which caused `condition`.

### (cell-error-name condition)
- Name of cell which caused `condition`.

### (unbound-slot-instance condition)
- Instance with unbound slot which caused `condition`.

### (print-not-readable-object condition)
- The object not readable printable under `condition`.

### (package-error-package condition)
- Package, path, or stream, respectively, which caused the condition of indicated `type`.

### (type-error-datum condition)
- Object which caused condition of type `type-error`, or its expected `type`, respectively.

### (simple-condition-format-control condition)
- Return `format-control` or list of `format` arguments, respectively, of `condition`.

### (break-on-signals)
- Condition type debugger is to be invoked on.
(make-condition condition-type (list:initialize-name value)*)
  ➤ Return new instance of condition-type.

(signal condition (condition-type (list:initialize-name value)*)
  ➤ Unless handled, signal as condition, warning or error, respectively, condition or a new instance of condition-type or with format control and args (see page 36), simple-condition, simple-warning, or simple-error, respectively. From signal and warn, return NIL.

(error continue-control (condition-concrete-arg (condition-type (list:initialize-name value)*
  ➤ Unless handled, signal as correctable error condition or a new instance of condition-type or, with format control and args (see page 36), error. In the debugger, use format arguments continue-control and continue-args to tag the continue option. Return NIL.

(ignore-errors form*)
  ➤ Return values of forms or, in case of errors, NIL and the condition.

(invoke-debugger condition)
  ➤ Invoke debugger with condition.

(assert text (condition-concrete-arg (condition-type (list:initialize-name value)*
  ➤ If text, which may depend on places, returns NIL, signal as correctable error condition or a new instance of condition-type or, with format control and args (see page 36), error. When using the debugger’s continue option, places can be altered before re-evaluation of test. Return NIL.

(handler-case foo (type (var)) (declare decl*) condition-form*)
  ➤ If, on evaluation of foo, a condition of type is signalled, evaluate matching condition-forms with var bound to the condition, and return their values. Without a condition, bind ord-As to values of foo and return values of forms or, without a no-error clause, return values of foo. See page 17 for (ord-lambda).

(handler-bind (condition-type handler-function*) form*)
  ➤ Return values of forms after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.

(with-simple-restart (restart) (condition-type handler-function*) form*)
  ➤ Return values of forms unless restart is called during their evaluation. In this case, describe restart using format control and args (see page 36) and return NIL and T.

(restart-case form (restart (ord-lambda)) (condition-type handler-function*)
  ➤ Return values of form or, if during evaluation of form one of the dynamically established restarts is called, the values of its restart-forms. A restart is visible under condition if (funcall #’test-function condition) returns T. If presented in the debugger, restarts are described by string or by #’report-function (of a stream). A restart can be called by (invoke-restart restart arg*), where args match ord-lambda, or by (invoke-restart-interactively restart) where a list of the respective args is supplied by #’arg-function. See page 17 for ord-lambda.

(allocate-instance class (list:initialize-value) other-keyarg*)
  ➤ Return uninitialized instance of class. Called by make-instance.

(shared-initialize instance (list:initialize-slots) (list:initialize-value) other-keyarg*)
  ➤ Fill the initial-slots of instance with the corresponding values, and fill those list:initialize-slots that are not list:initialize-value with the values of their list:initialize-forms.

(slot-missing class instance slot setf slot-boundp slot-makunbound value)

(slot-unbound class instance slot)
  ➤ Called on attempted access to non-existing or unbound slot. Default methods signal error/unbound-slot, respectively. Not to be called by user.

10.2 Generic Functions

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

(defgeneric foo (setf foo) (required-var* &optional (var* var))
  ➤ (argument-precedence-order required-var+) (document-citation string)
  ➤ (generic-function-class gf-class (declare gf-class (generalized superclass) method-class method-class) method-class)
  ➤ (method-combination c-type (declare c-type (generalized c-arg*) method-function-args))
  ➤ Define or modify generic function foo. Remove any methods previously defined by defgeneric. gf-class and the lambda parameters required-var* and var* must be compatible with existing methods. defmethod-args resemble those of defmethod. For c-type see section 10.3.

(ensure-generic foo (setf foo) (argument-precedence-order required-var+ (document-citation string)
  ➤ (generic-function-class gf-class (method-class method-class) method-class)
  ➤ (method-combination c-type c-arg* lambda-list lambda-list)
  ➤ (environment environment)
  ➤ Define or modify generic function foo. gf-class and lambda-list must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to method-class do not propagate to existing methods. For c-type see section 10.3.

(defmethod foo (setf foo) (before after)
  ➤ (primary-method var (spec-var class eql bar))
  ➤ (optional var (init supplied-p))
  ➤ (var (init supplied-p))
  ➤ (var (init supplied-p))
  ➤ (var (key var))
  ➤ (var (key var))
  ➤ (var (init))

  ➤ (declare decl*) (form*)
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` function. Applicable qualifiers depend on the method combination type; see section 10.3.

```
; define-method
  (add-method nocall-method generic-function-method)
  ; Add (if necessary) or remove (if any) method to/from
  ; generic-function.

; (find-method generic-function qualifiers specializers [error])
  ; Return suitable method, or signal error.

; (compute-applicable-methods generic-function args)
  ; List of methods suitable for args, most specific first.

; (call-next-method args current-pred arg)
  ; From within a method, call next method with args; return
  ; its values.

; (no-applicable-method generic-function any)
  ; Called on invocation of generic-function on args if there is
  ; no applicable method. Default method signals error. Not to
  ; be called by user.

; (invalid-method-error control arg)
  ; Signal error on applicable method with invalid qualifiers,
  ; or on method combination. For control and args see format,
  ; page 36.

; (no-next-method generic-function method arg)
  ; Called on invocation of call-next-method when there is no
  ; next method. Default method signals error. Not to be called
  ; by user.

; (function-keywords method)
  ; Return list of keyword parameters of method and T if other
  ; keys are allowed.

; (method-qualifiers method)
  ; List of qualifiers of method.
```

### 10.3 Method Combination Types

#### standard
- Evaluate most specific `around` method supplying the values of the generic function. From within this method, `call-next-method` can call less specific `around` methods if there are any. If not, or if there are no `around` methods at all, call all `before` methods, most specific first, and the most specific primary method which supplies the values of the calling `call-next-method` if any, 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.

<table>
<thead>
<tr>
<th>C-TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>most-specific-first</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>most-specific-last</td>
<td>Method with the least specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>most-specific-first</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>most-specific-last</td>
<td>Method with the least specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>primary-method</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>generic-function</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>condition</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>unexpected</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
<tr>
<td>error</td>
<td>Method with the most specific <code>around</code> methods that supply values.</td>
</tr>
</tbody>
</table>

- Simple built-in method combination types; have the same usage as the `c-types` defined by the short form of `define-method-combination`.

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

```
; define-method-combination
  (define-method-combination c-type (ord-λ*) (group
    | (qualifier [⋆])
      | (method-selection control)
      | (order most-specific-first)
      | (required bool)
      | (arguments method-combination-λ*)
      | (generic-function symbol)
      | (declare decl)**
      | (body body*)
  )

; call-method
  (call-method
    | (make-method form)
    | (next-method (method-combination body))
  )
```

- 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 31.

```
; define-condition
  (define-condition foo (parent-type) (condition
    | (slot (reader reader)*)
    | (writer writer)
    | (accessor accessor)*)
  )
```

- Define, as a subtype of `parent-types`, condition type `foo`. In a new condition, a slot’s value defaults to `form` unless set via `:initarg`/`initarg-name`; it is readable via `(reader i)` or `(accessor i)`, and writable via `(writer value i)` or `(accessor value i)` value.

With `allocation class`, `slot` is shared by all conditions of type `foo`. A condition is reported by `string` or by `report-function` of arguments `condition` and `stream`.

26