Spice Lisp User's Guide

Edited by Scott E. Fahlman and Monica J. Cellio

9 November 1983


Copyright © 1983 Carnegie-Mellon University

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
Chapter 1

Introduction

Common Lisp is a new dialect of Lisp, closely related to Maclisp and Lisp Machine Lisp. Common Lisp was developed in response to the need for a modern, stable, well documented dialect of Lisp that can be implemented efficiently on a variety of machine architectures.

Spice Lisp is the implementation of Common Lisp for microcodable personal machines running CMU's Spice computing environment. At present, Spice runs only on the Three Rivers Computer Corporation's PERQ; implementations for other machines are planned but not yet under way. Compatible versions of Common Lisp will soon be available for the DEC Vax, under both VMS and Unix, the Decsystem-20 with extended addressing, and the Symbolics 3600.

The central document for users of any Common Lisp implementation is the Common Lisp Reference Manual, by Guy L. Steele Jr. All implementations of Common Lisp must conform to this standard. However, a number of design choices are left up to the implementor, and implementations are free to add to the basic Common Lisp facilities. This document covers those choices and features that are specific to the Spice Lisp implementation. The Common Lisp Reference Manual and Spice Lisp User's Guide, taken together, should provide everything that the user of Spice Lisp needs to know.

For now, a number of documents describing useful library modules that run in Spice Lisp are included here. Once there are enough of these, the documents will be moved into a separate document on the Spice Lisp Program Library.

Spice Lisp is currently undergoing intensive tuning and development. For the next year or so, at least, new releases will be appearing frequently. This document will be modified for each major release, so that it is always up to date. Users of Spice Lisp at CMU should watch the SPICE and CLISP bulletin boards for release announcements, pointers to updated documentation files, and other information of interest to the user community.

1.1. Obtaining and Running Spice Lisp

In order to run Spice Lisp, you must have a Perq 1a or Perq 2 with 16K control store. You must also have an up-to-date Accent system. Use the update program to get the current release of Accent. Then, decide
where you want the Spice Lisp files to live. There must be at least 3500 pages free in the partition you wish to put Spice Lisp in. It is suggested that you make a subdirectory called slisp in the user partition. Path to the directory you want to put Spice Lisp in, then run the update program on the directory /usr/spice/slisp/run. When Spice Lisp is on your Perq, put the directory that it resides in on your search list and then just type lisp to the Accent shell.
Chapter 2
Implementation Dependent Design Choices

Several design choices in Common Lisp are left to the individual implementation. This chapter contains a partial list of these topics and the choices that are implemented in Spice Lisp.

2.1. Numbers

Currently, short-floats and single-floats are the same, and long-floats and double-floats are the same. Short floats use an immediate (non-consing) representation with 8 bits of exponent and a 21-bit mantissa. Long floats are 64-bit consed objects, with 12 bits of exponent and 53 bits of mantissa. All of these figures include the sign bit and, for the mantissa, the "hidden bit". The long-float representation conforms to the 64-bit IEEE standard, except that we do not support all the exceptions, negative 0, infinities, and the like.

Fixnums are stored as 28-bit two's complement integers, including the sign bit. The most positive fixnum is $2^{27} - 1$, and the most negative fixnum is $-2^{27}$. An integer outside of this range is a bignum.

2.2. Characters

Spice Lisp characters have 8 bits of code, 8 bits of font, and 8 control bits. The font bits are not used, and only 4 of the control bits are used: (control, meta, super, and hyper).

The control bit functions Control, Meta, Super, and Hyper are defined as in the Common Lisp Manual. The Perq keyboard does not produce these and Accent does not pass them to Spice Lisp, but programs can use these internally.

2.3. Vector Initialization

If no initial-value is specified, vectors of Lisp objects are initialized to nil, and vectors of integers are initialized to 0.
2.4. Packages

Common Lisp requires four built-in packages: **lisp**, **user**, **keyword**, and **system**. In addition to these, Spice Lisp has separate packages for **hemlock** (the editor) and **compiler**.

2.5. The Editor

The `ed` function will invoke the Hemlock Editor.

2.6. Time Functions

There are at present no time functions in Spice Lisp, due to the difficulty of getting such information from Accent. This is being worked on.

2.7. System Dependent Constants

The following constants are defined in Spice Lisp.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boole-clr</td>
<td>0</td>
</tr>
<tr>
<td>boole-set</td>
<td>1</td>
</tr>
<tr>
<td>boole-1</td>
<td>2</td>
</tr>
<tr>
<td>boole-2</td>
<td>3</td>
</tr>
<tr>
<td>boole-c1</td>
<td>4</td>
</tr>
<tr>
<td>boole-c2</td>
<td>5</td>
</tr>
<tr>
<td>boole-and</td>
<td>6</td>
</tr>
<tr>
<td>boole-ior</td>
<td>7</td>
</tr>
<tr>
<td>boole-xor</td>
<td>8</td>
</tr>
<tr>
<td>boole-eqv</td>
<td>9</td>
</tr>
<tr>
<td>boole-nand</td>
<td>10</td>
</tr>
<tr>
<td>boole-nor</td>
<td>11</td>
</tr>
<tr>
<td>boole-andc1</td>
<td>12</td>
</tr>
<tr>
<td>boole-andc2</td>
<td>13</td>
</tr>
<tr>
<td>boole-orc1</td>
<td>14</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>-------</td>
</tr>
<tr>
<td>boole-orc2</td>
<td>15</td>
</tr>
<tr>
<td>most-positive-fixnum</td>
<td>134217727</td>
</tr>
<tr>
<td>most-negative-fixnum</td>
<td>-134217728</td>
</tr>
<tr>
<td>most-positive-short-float</td>
<td>1.7014e38</td>
</tr>
<tr>
<td>least-positive-short-float</td>
<td>0.0</td>
</tr>
<tr>
<td>least-negative-short-float</td>
<td>0.0</td>
</tr>
<tr>
<td>most-negative-short-float</td>
<td>-1.7014e38</td>
</tr>
<tr>
<td>most-positive-single-float</td>
<td>1.7014e38</td>
</tr>
<tr>
<td>least-positive-single-float</td>
<td>0.0</td>
</tr>
<tr>
<td>least-negative-single-float</td>
<td>0.0</td>
</tr>
<tr>
<td>most-negative-single-float</td>
<td>-8.5071e38</td>
</tr>
<tr>
<td>most-positive-double-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>least-positive-double-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>least-negative-double-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>most-negative-double-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>most-positive-long-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>least-positive-long-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>least-negative-long-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>most-negative-long-float</td>
<td>0.0d0</td>
</tr>
<tr>
<td>short-float-epsilon</td>
<td>4.76837e-7</td>
</tr>
<tr>
<td>single-float-epsilon</td>
<td>4.76837e-7</td>
</tr>
<tr>
<td>double-float-epsilon</td>
<td>0.0d0</td>
</tr>
<tr>
<td>long-float-epsilon</td>
<td>0.0d0</td>
</tr>
<tr>
<td>short-float-negative-epsilon</td>
<td>4.76837e-7</td>
</tr>
<tr>
<td>single-float-negative-epsilon</td>
<td>4.76837e-7</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>double-float-negative-epsilon</td>
<td>0.0d0</td>
</tr>
<tr>
<td>long-float-negative-epsilon</td>
<td>0.0d0</td>
</tr>
<tr>
<td>char-code-limit</td>
<td>256</td>
</tr>
<tr>
<td>char-font-limit</td>
<td>256</td>
</tr>
<tr>
<td>char-bits-limit</td>
<td>256</td>
</tr>
<tr>
<td>char-control-bit</td>
<td>1</td>
</tr>
<tr>
<td>char-meta-bit</td>
<td>2</td>
</tr>
<tr>
<td>char-super-bit</td>
<td>4</td>
</tr>
<tr>
<td>char-hyper-bit</td>
<td>8</td>
</tr>
<tr>
<td>array-rank-limit</td>
<td>134217727</td>
</tr>
<tr>
<td>internal-time-units-per-second</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 3
Debugging Tools

By Jim Large

3.1. The Break Loop

The break loop is a read-eval-print loop which is similar to the normal lisp top level. It can be called from any lisp function to allow the user to interact with the lisp system. When the user gives the command to exit the break loop, he may choose an arbitrary value for it to return.

When a lisp expression is typed in at the break loop's prompt, it is usually evaluated and printed. However, there are three special expressions which are recognized as break loop commands, and which are not evaluated.

$G
Typing this symbol causes a throw to the lisp top level: The current computation is aborted, and all bindings are unwound.

$P
Typing this symbol causes the break loop to return nil.

(RETURN form) Typing this expression causes the break loop to evaluate form and return the result(s).

The dollar sign character in the symbols $P and $G is intended to be the (escape) character -- ascii 27. For compatibility with the VAX VMS operating system, real dollar signs will be recognized also.

When the break loop is called, it tries to make sure that terminal interaction will be possible. All of the standard input output streams, *standard-input*, *standard-output*, *error-output*, *query-io*, and *trace-output* are bound to *terminal-io* for the duration of the break loop; and the state of the single stepper is bound to "off".

break tag &optional condition [Macro]
The break macro returns a form which prints the message "Breakpoint tag" to *terminal-io* and then invokes the break loop. If condition is present, then the form evaluates it and tests the result. If the result is nil, then the form returns nil; otherwise, the form prints the tag and invokes the break loop. tag is never evaluated.
3.1.1. Cleaning Up

The break loop is called by the system error handlers. Since errors can happen unexpectedly, the break loop provides a mechanism for cleaning up any unusual state that a program may have caused.

*error-cleanup-forms*  [Variable]

A list of lisp forms which will be evaluated for side effect when a break loop is invoked. Whenever a break loop is entered, *error-cleanup-forms* will be bound to nil, and then the forms which were its previous value will be evald for side effect. There is no way to have the side effects undone when the break loop returns, and if any of the cleanup forms causes an error, the result can not be guaranteed.

As an example, a program that puts the terminal in an unusual mode might want to do something like this.

\[
\text{(let } \{ (*error-cleanup-forms*}
\text{   (cons '}(\text{progn <code to restore terminal>}}
\text{       *error-cleanup-forms*)\text{))}
\text{   <code to mess up terminal>}
\text{   ...}}
\]

3.2. Function Tracing

The tracer causes selected functions to print their arguments and their results whenever they are called. Options allow conditional printing of the trace information and conditional breakpoints on function entry.

trace &rest specs  [Macro]

Invokes tracing on the specified functions, and pushes their names onto the global list in *traced-function-list*. Each spec is either the name of a function, or the form

\[
(function-name
   \text{trace-option-name value}
   \text{trace-option-name value}
   \ldots)
\]

If no specs are given, then trace will return the list of all currently traced functions, *traced-function-list*.

If a function is traced with no options, then each time it is called, a single line containing the name of the function, the arguments to the call, and the depth of the call will be printed on the stream *trace-output*. After it returns, another line will be printed which contains the depth of the call and all of the return values. The lines are indented to highlight the depth of the calls.

Trace options can cause the normal printout to be suppressed, or cause extra information to be printed. Each traced function carries its own set of options which is independent of the options

---

1 Trace does not work on macros or special forms yet.
given for any other function. Every time a function is specified in a call to trace, all of the old options are discarded. The available options are:

:condition  A form to eval before each call to the function. Trace printout will be suppressed whenever the form returns nil.

:break  A form to eval before each call to the function. If the form returns non nil, then a breakpoint loop will be entered immediately before the function call.

:break-after  Like :break, but the form is evaled and the break loop invoked after the function call.

:break-all  A form which should be used as both the :break and the :break-after args.

:wherein  A function name or a list of function names. Trace printout for the traced function will only occur when it is called from within a call to one of the :wherein functions.

:print  A list of forms which will be evaluated and printed whenever the function is called. The values are printed one per line, and indented to match the other trace output. This printout will be suppressed whenever the normal trace printout is suppressed.

:print-after  Like :print except that the values of the forms are printed whenever the function exits.

:print-all  The arg is used as the arg to both :print and :print-after.

untrace &rest function-names [Macro]

Turns off tracing for the specified functions, and removes their names from *traced-function-list*. If no function-names are given, then all functions named in *traced-function-list* will be untraced.

*traced-function-list* [Variable]

A list of function names which is maintained and used by trace, untrace, and untrace-all. This list should contain the names of all functions which are currently being traced.

*trace-prinlevel* [Variable]

*trace-prinlength* [Variable]

*Prinlevel* and *prinlength* are bound to *trace-prinlevel* and *trace-prinlength* when printing trace output. The forms printed by the :print options are affected also. *Trace-prinlevel* and *trace-prinlength* are initially set to nil.

*max-trace-indentation* [Variable]

The maximum number of spaces which should be used to indent trace printout. This variable is initially set to some reasonable value.
3.2.1. Encapsulation Functions

The encapsulation functions provide a clean mechanism for intercepting the arguments and results of a function. Encapsulate changes the function definition of a symbol, and saves it so that it can be restored later. The new definition normally calls the original definition.

The original definition of the symbol can be restored at any time by the unencapsulate function. Encapsulate and unencapsulate allow a symbol to be multiply encapsulated in such a way that different encapsulations can be completely transparent to each other.

Each encapsulation has a type which may be an arbitrary lisp object. If a symbol has several encapsulations of different types, then any one of them can be removed without affecting more recent ones. A symbol may have more than one encapsulation of the same type, but only the most recent one can be undone.

\texttt{encapsulate symbol type body} \hspace{1cm} \textit{[Function]}

Saves the current definition of \texttt{symbol}, and replaces it with a function which returns the result of evaluating the form, \texttt{body}. \texttt{Type} is an arbitrary lisp object which is the type of encapsulation.

When the new function is called, the following variables will be bound for the evaluation of \texttt{body}:

\texttt{argument-list}

A list of the arguments to the function.

\texttt{basic-definition}

The unencapsulated definition of the function.

The unencapsulated definition may be called with the original arguments by including the form

\begin{verbatim}
(apply basic-definition argument-list)
\end{verbatim}

Encapsulate always returns \texttt{symbol}.

\texttt{unencapsulate symbol type} \hspace{1cm} \textit{[Function]}

Undoes \texttt{symbol}'s most recent encapsulation of type \texttt{type}. \texttt{Type} is compared with \texttt{eq}. Encapsulations of other types are left in place.

\texttt{encapsulated-p symbol type} \hspace{1cm} \textit{[Function]}

Returns \texttt{t} if \texttt{symbol} has an encapsulation of type \texttt{type}. Returns \texttt{nil} otherwise. \texttt{Type} is compared with \texttt{eq}.

\footnote{Encapsulation does not work for macros or special forms yet.}
3.3. Single Stepper

\textbf{step form} \quad \textbf{[Function]}

Evaluates form with single stepping enabled, or if form is t, enables stepping until explicitly disabled. Stepping can be disabled by quitting to the lisp top level, or by evaluating the form (\texttt{step ()}).

While stepping is enabled, every call to \texttt{eval} will prompt the user for a single character command. The prompt is the form which is about to be evaluated. It is printed with \texttt{*prinlevel*} and \texttt{*prinlength*} bound to \texttt{*step-prinlevel*} and \texttt{*step-prinlength*}. All interaction is done through the stream \texttt{*query-io*}.

The commands are:

\begin{itemize}
  \item \texttt{n (next)} \quad \text{Evaluate the expression with stepping still enabled.}
  \item \texttt{s (skip)} \quad \text{Evaluate the expression with stepping disabled.}
  \item \texttt{q (quit)} \quad \text{Evaluate the expression, but disable all further stepping inside the current call to step.}
  \item \texttt{p (print)} \quad \text{Print current form. (does not use \texttt{*step-prinlevel*} or \texttt{*step-prinlength*.})}
  \item \texttt{b (break)} \quad \text{Enter break loop, and then prompt for the command again when the break loop returns.}
  \item \texttt{e (eval)} \quad \text{Prompt for and evaluate an arbitrary expression. The expression is evaluated with stepping disabled.}
  \item \texttt{? (help)} \quad \text{Prints a brief list of the commands.}
  \item \texttt{r (return)} \quad \text{Prompt for an arbitrary value to return as result of current call to eval.}
  \item \texttt{g} \quad \text{Throw to top level.}
\end{itemize}

\texttt{*step-prinlevel*} \quad \textbf{[Variable]}

\texttt{*step-prinlength*} \quad \textbf{[Variable]}

\texttt{*prinlevel*} and \texttt{*prinlength*} are bound to these values when the current form is printed.

\texttt{*step-prinlevel*} and \texttt{*step-prinlength*} are initially bound to some small value.

\texttt{*max-step-indentation*} \quad \textbf{[Variable]}

Step indents the prompts to highlight the nesting of the evaluation. This variable contains the maximum number of spaces to use for indenting. It is initially set to some reasonable number.

3.4. The Debugger

The debugger is an interactive command loop which allows a user to examine the active call frames on the Lisp function call stack. If it is invoked from an error breakpoint, it can show the function calls which led up
to the error.

Only one stack frame, the current frame, can be inspected at any given time. The command loop prints the frame number of the current frame as a prompt, and then reads a lisp expression from the terminal. debug tries to use the expression as a command, but if it fails, then it evals and prints the expression like a breakpoint loop. Terminal input and output are done by binding *standard-input* and *standard-output* to *terminal-io*.

3.4.1. Movement Commands

These commands move to a new stack frame, and print out the name of the function and the values of its arguments in the style of a lisp function call. *debug-prinlevel* and *debug-prinlength* affect the style of the printing.

Up is toward the most recent function invocation, and lower frame numbers. Down is toward older function calls and higher frame numbers.

Visible frames are those which have not been hidden by the hide command which is described below. The special variable *debug-ignored-functions* contains a list of function names which are hidden by default.

The commands are:

\[ \begin{align*}
\texttt{u} & \quad \text{Move up to the next higher visible frame.} \\
\texttt{d} & \quad \text{Move down to the next lower visible frame.} \\
\texttt{t} & \quad \text{Move to the highest visible frame.} \\
\texttt{b} & \quad \text{Move to the lowest visible frame.} \\
\texttt{(frame n)} & \quad \text{Move to frame number } n \text{ whether it is visible or not.}
\end{align*} \]

3.4.2. Inspection Commands

These commands print information about the current frame and the current function.

\[ \begin{align*}
\texttt{?} & \quad \text{describe’s the current function.} \\
\texttt{a} & \quad \text{Lists the arguments to the current function. The values of the arguments are printed along with the argument names.} \\
\texttt{l} & \quad \text{Lists the local variables in the current function. The values of the locals are printed, but their names are no longer available.} \\
\texttt{p} & \quad \text{Redisplays the current function call as it would be displayed by moving to this frame.}
\end{align*} \]
DEBUGGING TOOLS

**pp** Redisplays the current function call using *prinlevel* and *prinlength* instead of *debug-prinlevel* and *debug-prinlength*.

**(value symbol)** Prints the value of symbol in the current binding context. symbol is either a special variable, or the name of an argument to the current function.

**(local n)** Prints the value of the nth local variable in the current frame.

### 3.4.3. Other Commands

**h** Prints a brief list of commands on the terminal.

**q** Causes debug to return nil.

**(return &rest values)** Forces the current function to return zero or more values. If the function was not called for multiple values, then attempts to return other than one value will be prevented.

**(backtrace options)** Prints a history of function calls. The printing is controlled by *debug-prinlevel* and *debug-prinlength*. Only those frames which are considered visible by the frame movement commands will be shown. Currently, there are no options.

**(hide options)** Makes the stack frames described by options be invisible to the frame movement commands. The first option is a subcommand which may be one of:

- **package** Followed by one or more package names. Calls to functions in the named packages will not be visible.

- **function** Followed by one or more function names. Calls to the named functions will not be visible.

- **compiled** Calls to compiled functions will not be visible.

- **interpreted** Calls to interpreted functions will not be visible.

- **lambdas** Calls to lambda expressions will not be visible.

**(show options)** Options are the same as for the hide command. Show is the inverse operation. Showing a particular set of functions does not necessarily make them all visible; some of them may still be hidden for other reasons.

**debug** [Function] Invokes the debugger. Always returns nil.
*debug-prinlevel*  [Variable]
*debug-prinlength*  [Variable]

*Prinlevel* and *prinlength* are bound to these values during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal *Prinlevel* and *prinlength* are in effect. These variables are initially set to some small number.

*debug-ignored-functions*  [Variable]

A list of functions which are hidden by default. These functions can be made visible with the debug command show.

3.5. Random Features

describe  [Function]

The describe function prints useful information about object on *standard-output*. For any object, describe will print out the type. Then it prints other information based on the type of object. The types which are presently handled are:

function  describe prints a list of the function's name (if any) and its formal parameters. If the function has documentation, then the documentation string will be printed.

symbol  The symbol's value, properties, and documentation are all printed. If the symbol has a function definition, then the function is described.
Chapter 4

The Compiler

4.1. Calling the Compiler

Functions are compiled using either the compile or compile-file functions. Both operate as documented in the Common Lisp Reference Manual. The compile-file function takes the following keyword arguments: :output-file, :clog-file, :lap-file, and :error-file. These take either the name of a file, as a string, or the symbol t, which tells the system to make up an appropriate name by replacing the type field of the input file name. If any of these arguments is NIL, no output of that type is created. By default, only the output file and error file are created.

4.2. Open and Closed Coding

When a function call is "open coded," inline code whose effect is equivalent to the function call is substituted for that function call. When a function call is "closed coded", it is usually left as is, although it might be turned into a call to a different function with different arguments. As an example, if nthcdr were to be "open coded" then

(nthcdr 4 foobar)

might turn into

(cdr (cdr (cdr (cdr foobar))))

or even

(do ((i 0 (1+ i))
    (list foobar (cdr foobar)))
   ((= i 4) list)).

If nth is "closed coded"

(nth x 1)

might stay the same, or turn into something like:

(car (nthcdr x 1)).
4.3. Compiler Switches

Several compiler switches are available which are not documented in the Common Lisp Manual. Each is a global special. These are described below.

*peep-enable* If this switch is non-nil, the compiler runs the peephole optimizer. The optimizer makes the compiled code faster, but the compilation itself is slower. *peep-enable* defaults to t.

*peep-statistics* If this switch is non-nil, the effectiveness of the peephole optimizer (number of bytes before and after optimization) will be reported as each function is compiled. *peep-statistics* defaults to t.

*inline-enable* If this switch is non-nil, then functions which are declared to be inline are expanded inline. It is sometimes useful to turn this switch off when debugging. *inline-enable* defaults to t.

*open-code-sequence-functions* If this switch is non-nil, the compiler tries to translate calls to sequence functions into do loops, which are more efficient. It defaults to t.

*nthcdr-open-code-limit* This is the maximum size an nthcdr can be to be open coded. In other words, if nthcdr is called with n equal to some constant less than or equal to the *nthcdr-open-code-limit*, it will be open coded as a series of nested cdr's. *nthcdr-open-code-limit* defaults to 10.

*complain-about-inefficiency* If this switch is non-nil, the compiler will print a message when certain things must be done in an inefficient manner because of lack of declarations or other problems of which the user might be unaware. This defaults to nil.

*eliminate-tail-recursion* If this switch is non-nil, the compiler attempts to turn tail recursive calls (from a function to itself) into recursion. This defaults to t.

*all-rest-args-are-lists* If non-nil, this has the effect of declaring every &rest arg to be of type list. (They all start that way, but the user could alter them.) It defaults to nil.

*verbose* If this switch is nil, only true error messages and warnings go to the error stream. If non-nil, the compiler prints a message as each function is compiled. It defaults to t.

*check-keywords-at-runtime* If non-nil, compiled code with &key arguments will check at runtime for unknown keywords. This is usually left on and defaults to t.
4.4. Declare switches

Not all switches for declare are processed by the compiler. The ftype and function declarations are currently ignored.

The optimize declaration controls some of the above switches:

- *peep-enable* is on unless csped is greater than speed and space.
- *inline-enable* is on unless space is greater than speed.
- *open-code-sequence-functions* is on unless space is greater than speed.
- *eliminate-tail-recursion* is on if speed is greater than space.
Chapter 5

Efficiency

By Rob McIlrath

In Spice Lisp on the Perq, as is any language on any computer, the way to get efficient code is to use good algorithms and sensible programming techniques, but to get the last bit of speed it is helpful to know some things about the language and its implementation. This chapter is a summary of various hidden costs in the implementation and ways to get around them.

5.1. Compile Your Code

In Spice Lisp, compiled code typically runs at least sixty times faster than interpreted code. Another benefit of compiling is that it catches many typos and other minor programming errors. Many Lisp programmers find that the best way to debug a program is to compile the program to catch simple errors, then debug the code interpreted, only actually using the compiled code once the program is debugged.

Another benefit of compilation is that compiled (.sfl) files load significantly faster, so it is worthwhile compiling files which are loaded many times even if the speed of the functions in the file is unimportant.

*Do Not* be concerned about the performance of your program until you see its speed compiled -- some techniques that make compiled code run faster make interpreted code run slower.

5.2. Avoid Unnecessary Consing

Consing is the Lispy name for allocation of storage, as done by the cons function, hence its name. cons is by no means the only function which conses -- so does make-array and many other functions. Even worse, the Lisp system may decide to cons furiously when you do some apparently innocent thing.

Consing hurts performance in the following ways:

- Consing reduces your program’s memory access locality, increasing paging activity.
- Consing takes time just like anything else.
- Any space allocated eventually needs to be reclaimed, either by garbage collection or killing your
Lisp.

Of course you have to cons sometimes, and the Lisp implementors have gone to considerable trouble to make consing and the subsequent garbage collection as efficient as possible. In some cases strategic consing can improve speed. It would certainly save time to allocate a vector to store intermediate results which are used hundreds of times.

5.3. Do, Don’t Map

One of the programming styles encouraged by Lisp is a highly applicative one, involving the use of mapping functions and many lists to store intermediate results. To compute the sum of the square-roots of a list of numbers, one might say:

\[
\text{apply } \#'+ (\text{mapcar } \#'\text{sqrt list-of-numbers})
\]

This programming style is clear and elegant, but unfortunately results in slow code. There are two reasons why:

- The creation of lists of intermediate results causes much consing (see 5.2).
- Each level of application requires another scan down the list. Thus, disregarding other effects, the above code would probably take twice as long as a straightforward iterative version.

An example of an iterative version of the same code:

\[
\text{do} ((\text{num list-of-numbers (cdr num)})
\qquad (\text{sum 0 (+ (sqrt (car num)) sum)})
\qquad ((\text{null num) sum}))
\]

Once you feel in you heart of hearts that iterative Lisp is beautiful then you can join the ranks of the Lisp efficiency fiends.

5.4. Think Before You Use a List

Although Lisp’s creator seemed to think that it was for LIst Processing, the astute observer may have noticed that the chapter on list manipulation makes up less that ten percent of the COMMON LISP manual. The language has grown since Lisp 1.5, and now has other data structures which may be better suited to tasks where lists might have been used before.

5.4.1. Use Vectors

*Use Vectors* and use them often. Lists are often used to represent sequences, but for this purpose vectors have the following advantages:

- A vector takes up less space than a list holding the same number of elements. The advantage may vary from a factor of two for a general vector to a factor of sixty-four for a bit-vector. Less space means less consing (see 5.2).
- Vectors allow constant time random-access. You can get any element out of a vector as fast as you
can get the first out of a list if you make the right declarations.

The only advantage that lists have over vectors for representing sequences is that it is easy to change the length of a list, add to it and remove items from it. Likely signs of archaic, slow lisp code are nth and nthcdr -- if you are using these function you should probably be using a vector.

5.4.2. Use Structures

Another thing that lists have been used for is the representation of record structures. Often the structure of the list is never explicitly stated and accessing macros are not used, resulting in impenetrable code such as:

\[
\text{replace (caddr (cadddr x)) (caddr y)}
\]

The use of defstruct structures can result in much clearer code, one might write instead:

\[
\text{setf (beverage-flavor (astronaut-beverage x)) (beverage-flavor y)}
\]

*Great!* But what does this have to do with efficiency? Since structures are based on vectors, the defstruct version would likewise take up less space and be faster to access. Don't be tempted to try and gain speed by trying to use vectors directly, since the compiler knows how to compile faster accesses to structures than you could easily do yourself. Note that the structure definition should be compiled before any uses of accessors so that the compiler will know about them.

5.4.3. Use Hashables

In many applications where association lists (alists) have been used in the past, hashtables would work much better. An alist may be preferable in cases where the user wishes to rebind the alist and add new values to the front, shadowing older associations. In most other cases, if an alist contains more than a few elements, a hashtable will probably do the job faster. If the keys in the hashtable are objects that can be compared with eql or better yet eq, then hashtable access will be speeded up by specifying the correct function as the :test argument to make-hashtable.

5.4.4. Use Bit-Vectors

Another thing that lists have been used for is set manipulation. In some applications where there is a known, reasonably small universe of items Bit-Vectors could be used instead. This is much less convenient than using lists, because instead of symbols, each element in the universe must be assigned a numeric index into the bit vector. If the universe is very small -- twenty-eight items or less -- then you can represent your set as bits in a fixnum and use logior and so on, to get immense speed improvements.

Note: right now, boolean operations on bit-vectors are very slow, since one bit is processed at a time instead of 16 or 32 bits at once. This will be fixed soon. In the meantime, boolean operations on bignums are faster than those on bit-vectors.
5.5. Simple Vs Complex Arrays

Spice Lisp has two different representations for arrays, one which is accessed rapidly in microcode and one which is accessed much more slowly in Lisp code. The class of arrays which can be represented in the fast form corresponds exactly to the *one dimensional simple-arrays*, as defined in the COMMON LISP manual. Included in this group are the types *simple-string*, *simple-vector* and *simple-bit-vector*.

*Declare Your Vector Variables* -- If you don't the compiler will be forced to assume you are using the inefficient form of vector. Example:

```lisp
(defun iota (n)
  (let ((res (make-vector n)))
    (declare (simple-vector n))
    (do*( i n)
      (setf (aref res i) i)
    res))
```

Warning: if you declare things to be simple when they are not, incorrect code will be generated and hard-to-find bugs will result. It is worthwhile to note, however, that system functions which create vectors will always create simple-arrays unless you force them to do otherwise.

5.6. To Call or Not To Call

The usual Lisp style involves small functions and many function calls; for this reason Lisp implementations strive to make function calling as inexpensive as possible. Spice Lisp is fairly successful in this respect. Function calling is not vastly more expensive than other instructions, and is certainly faster than procedure calling in Perq Pascal.

For this reason you should not be overly concerned about function-call overhead in your programs. *However*, function calling does take time, and thus is not the kind of thing you want going on in the inner loops of your program. Where removing function calling is desirable you can use the following techniques:

*Write the code in-line*

This is not a very good idea, since it results in obscure code, and spreads the code for a single logical function out everywhere, making changes difficult.

*Use macros*

A macro can be used to achieve the effect of a function call without the function-call overhead, but the extreme generality of the macro mechanism makes them tricky to use. If macros are used in this fashion without some care, obscure bugs can result.

*Use inline functions*

This often the best way to remove function call overhead in COMMON LISP. A function may be written, and then declared inline if it is found that function call overhead is excessive. Writing functions is easier that writing macros, and it is easier to declare a function inline than to convert it to a macro. Note that the compiler must process first the inline declaration, then the definition, and finally any calls which are to be open coded for the inline expansion to take place.
Note that any of the above techniques can result in bloated code, since they have the effect of duplicating the same instructions many places. If code becomes very large, paging may increase, resulting in a significant slowdown. Inline expansion should only be used where it is needed. Note that the same function may be called normally in some places and expanded inline in other places.

5.7. Keywords and the Rest

*COMMON LISP* has very powerful argument passing mechanisms. Unfortunately, two of the most powerful mechanisms, rest arguments and keyword arguments, have a serious performance penalty in Spice Lisp.

The main problem with rest args is that the microcode must cons a list to hold the arguments. If a function is called many times or with many arguments, large amounts of consing may occur.

Keyword arguments are built on top of the rest arg mechanism, and so have all the above problems plus the problem that a significant amount of time is spent parsing the list of keywords and values on each function call.

Neither problem is serious unless thousands of calls are being made to the function in question, so the use of argument keywords and rest args is encouraged in user interface functions.

Another way to avoid keyword and rest-arg overhead is to use a macro instead of a function, since the rest-arg and keyword overhead happens at compile time and not necessarily at runtime. If the macro-expanded form contains no keyword or rest arguments, then it is perfectly acceptable to use keywords and rest-args in macros which appear in inner loops.

Note: the compiler open-codes most heavily-used system functions which have keyword or rest arguments, so that no run-time overhead is involved.

Optional arguments have no significant overhead.

5.8. Numbers

*Spice Lisp* provides five types of numbers for your enjoyment:

- fixnums
- bignums
- ratios
- short-floats
- long-floats

Only short-floats and fixnums have an immediate representation; the rest must be consed and garbage-collected later. In code where speed is important, you should use only fixnums and short-floats unless you have a real need for something else. Since *most-positive-fixnum* is more than one hundred million, you shouldn't need to use bignums unless you are counting the reasons to use Lisp instead of Pascal. Unfortunately the amount of floating point precision that will fit in twenty-eight bits is severely limited, so there are reasonable problems which require the use of long-floats.
Another feature of ratios and bignums which will keep you entertained for hours is that operations on these numbers are written in Lisp, not microcode; this results in orders of magnitude slower execution.

Printing of long-floats is painfully slow -- around three seconds. While you wait, consider that the float printing algorithm is the only known correct float printing method. Other methods run in real time, but they lose precision in the low-order digits.

5.9. Timing

Everyone knows that the first step in improving a program's performance is to make extensive timings to find which code is time-critical. Unfortunately Spice Lisp currently has no timing functions. The recommended timing method is to write a compiled driver function which calls the function to be tested a few hundred times. If one measures the total time with a watch or by a systat and divides by the number of iterations, then fairly accurate statistics can be collected.
Chapter 6

The Alien Data Type

By Jim Large and Dan Aronson

A problem that arises in many Common Lisp implementations is dealing with the complex structured records or messages that are exchanged at the interface between Lisp and the outside world. Such alien data structures will typically be collections of integers, floating point numbers, strings, boolean flags, bit vectors, enumerated types represented as small integers, and so on. All of these types have some rough correspondence with internal Lisp data-types, but at the time of their arrival and departure they will be in whatever implementation-dependent format is expected by the alien software, and the Lisp garbage collector must treat the alien object as an unstructured vector of bits or bytes.

Given a knowledge of the structure of an alien record, it is relatively easy for the Lisp-level code to convert each field of the message into the corresponding Lisp form, but we want this knowledge to be concentrated in one place so that changes in the external message format can be easily accommodated by the Lisp code. What is needed is a convenient form for specifying how the alien record is to be parsed and packed and how each of its fields is to be interpreted as a Lisp object. In a manner similar to defstruct, this specification will be processed to create a family of field-accessing and field-altering macros that perform the proper translations, in addition to doing the access. Thus, once the structure of the alien record has been specified, it is no harder to access than the fields of a defstruct.

This new facility is built into Vax Common Lisp and Spice Lisp.

6.1. The Alien-Structure Data Type

There is a new data-type called alien-structure. This is just a new structure-type defined by defstruct. The alien structure contains a name (a lisp symbol), a length (number of 8-bit bytes in the data vector), and a pointer to the actual blob of uninterpreted bits. In Spice Lisp and Vax Common Lisp, this blob is an packed-fixnum vector (a U-Vector, in our internal parlance) of 8-bit bytes; other implementations might have to use a bit-vector for this. One could ask typep if an object is an alien-structure, and could access the innards via alien-structure-name and alien-structure-data. One can also find the length of the data area (in bytes) using the macro alien-structure-length.

Alien structures are created by a macro, def-alien-structure, that parallels defstruct in form:
(def-alien-structure (name option1 option2 ...)
  field-description-1
  field-description-2
  ...
)

where the field descriptions are of the form

(field-name alien-field-type start end
  field-option-name-1 field-option-value-1
  field-option-name-2 field-option-value-2
  ...
)

The options in def-alien-structure are a subset of those allowed in defstruct: :conc-name, :constructor, :predicate, :print-function, and :eval-when. Alien structures are always named and the user cannot specify a :type option or any of the array options.

In addition, there is a :length option that takes as its argument the length of the data area in 8-bit bytes. This is used when new instances of this structure-type are created within Lisp by the constructor macro. If :length is not specified, the length defaults to the maximum of the end values of the fields making up the structure, ignoring any fields with end values of nil. When alien structures are read in from outside the Lisp, :length controls the allocated length of the data vector, but the actual length (as reported by alien-structure-length) is set by the size of the incoming block of data. An error will be signalled if the incoming data block does not fit within the allocated length. If no :length is specified in this case, the default is to allocate a vector the same size as the incoming data block.

Each field has a name, perhaps modified by :conc-name. The field options are :invisible and :read-only, as in defstruct, plus

:default-value. The latter is a value that is placed into the slot at the time the alien structure is created, if no other value is specified at that time. This value is inserted into the alien structure as if by setf, and it is therefore processed by the field-type conversions (see below). If no :default-value is specified, the field is initialized to 0. (This initialization is only done if the data block is created within the Lisp; if it arrives from outside, the bits are left alone unless specifically altered by the user.)

The start and end values for a field indicate where, in the alien structure's data area, the field is to be found. These numbers are in terms of bytes. As usual, they are zero-based, start is inclusive and end is exclusive. If the end value is nil, the field has no fixed length, but runs from the specified start to the end of the data block, as indicated by alien-structure-length. When such a field is written into, the alien-structure-length will be adjusted to reflect the new end of the field; however, any attempt to extend the field beyond the allocated length of the data vector will signal an error.

It is possible for two fields to overlap; sometimes this will be useful when one field wants to be interpreted in two different ways. Obviously, if two overlapping fields are written into, the later write clobbers the results of the earlier one. It is also possible to have gaps between the defined fields; these would correspond to parts of an incoming message that are uninteresting to the Lisp program, for example. Such gaps are initialized to 0.
when the alien structure is created within the Lisp, unless the block of data comes in from outside. If the block comes from the outside, the bits in the inter-field gaps are not altered. Fields may appear in any order within **def-alien-structure**.

In the rare cases where the boundaries of a field do not land on byte-boundaries, rational numbers may be supplied as the **start** and **end** values. So most of the time you can pick up a string from (16 32) or an integer from (0 2), but sometimes you would get a boolean from (3/8 4/8) or an integer from (1/2 3/2). An error is signalled if the rational does not specify an integral bit-address.

The alien-field-type argument is a symbol that tells how the field is to be interpreted by the Lisp system. Each alien field type associates a particular alien-format representation with some internal Lisp data-type; functions exist for turning the contents of the field into the internal object and for packing an internal object of the right type back into the field. Some of these types will be built-in:

**string**

On access, the field is interpreted as a string, one character per byte, and the corresponding Lisp string is returned. The **setf** form accepts a Lisp string and puts the characters into the field.

**perq-string**

Like **string** except that the first byte in the field is the number of characters, and the remaining bytes are the characters. On access, the length of the Lisp string will be determined by the first byte of the field. The **setf** form will set the first byte according to the length of the Lisp string. The size of the field may not be greater than 256 bytes.

**signed-integer**

The field is interpreted as a signed, two's complement integer, and the corresponding Lisp integer is returned. On write, the process is reversed.

**unsigned-integer**

The field is interpreted as an unsigned, positive integer, and the corresponding Lisp integer is returned.

**bit-vector**

The field is returned as a bit-vector.

**port**

The field contains an Accent IPC port specified as a 32-bit integer. The internal Lisp format is a **port** structure, with the integer value hidden inside.

*(selection s0 s1 s2 ...)*

The Sn are evaluated (at access or **setf** time) to produce arbitrary Lisp objects. On access the alien field is interpreted as an unsigned integer, and the corresponding Sn value is returned inside the Lisp. On output, the setting function receives one of the values and stores the corresponding integer into the field. Comparison of items against Sn values is done with **eq1**.

**ieee-single-flonum**

The field is interpreted as containing a 32-bit IEEE-format flonum, and this is returned as the internal Lisp flonum type that most closely matches this type. This will vary from one implementation to another, but will be constant within a given implementation. An
implementation would provide whatever floating formats are important in its host environment -- the VAX might provide D, F, G, and H formats rather than IEEE formats.

In each case, the setf form will signal an error if the specified field is too small to hold the item coming from Lisp. For integers, the error will occur if significant bits would be lost in doing the write.

6.2. Defining Other Field Types

In addition to these built-in primitive alien field types, the user can define his own via def-alien-field-type, a macro with the following arguments:

\[
\text{def-alien-field-type name internal-type primitive-type access-fn setf-fn}
\]

\text{internal-type} is a Common Lisp type specifier indicating the type of internal Lisp object that the field will be mapped to. \text{primitive} is any pre-defined alien-field-type: one of the primitives defined above or a field type defined earlier by \text{def-alien-field-type}. On access, this is applied to the alien object to extract a Lisp object; then this object is passed to the access function, usually a function of one argument, for further processing. For a setf, the new value is first passed through the setf function, also usually a function of one argument; the result of this is then packed into the alien structure as indicated by \text{primitive-type}.

For example, suppose we wanted to create a new field-type named backwards-string, in which the alien field is treated as a reversed string. This would be done as follows:

\[
\text{(def-alien-field-type reverse-string 'string 'string #'(lambda (x) (reverse x)) #'(lambda (x) (reverse x)))}
\]

\text{Once this is done, the reverse-string field type can be used in def-alien-structure-type and as a primitive in defining still more complex field types.}

Sometimes, it is desirable to create an alien field type in which the access and setf conversion functions can take additional parameters. The selection field-type, listed above among the primitives, is one such type. To achieve this effect, one defines an alien field type whose access and setf functions take more than one argument. The additional arguments should be optional. When a field-type expression in def-alien-field is a list rather than a symbol, the car of the list is the type name, and the remaining elements are expressions which are evaluated at access and setf time. The results of these evaluations are passed to the access and setf functions as additional arguments. This all sounds more complex than it really is. To produce the selection field type, if this were not built in as a primitive, one would do the following:
(def-alien-field-type selection 't 'unsigned-integer ; Produces any kind of Lisp object.
  #(lambda (n &rest s-list)
      (nth n s-list)) ; Primitive access as unsigned integer.
  #(lambda (x &rest s-list)
      (position x s-list))) ; Select Nth value in list of choices.
  ; Find index of item in list, EQL test.

6.3. Variable-Format Structures

The machinery described above is optimized for dealing with alien structure types whose fields are fixed in size and position at the time the structure-type is defined. Given the nature of software outside the Lisp world, this is the sort of thing we will be seeing the most of. However, it would also be nice to be able to use the alien field type translation for packing and unpacking variable-format records. To handle this variable case without getting too complicated, the following simple packing and unpacking macros are provided:

(aliend-field alien-structure alien-field-type start end)

This form can be used to access an arbitrary field in any type of alien structure, using the specified alien-field-type, start, and end. This form pays no attention to any fixed-position fields that may have been defined for structures of this type; it just does what you tell it to do. The alien-field-type may be any pre-defined type. The start and end arguments are expressed in terms of 8-bit bytes, and may be ratios if it is necessary to reference a field that does not lie on even byte boundaries. This form may be used within a setf to alter a field.

(pack-alien-structure name length
  (value alien-field-type start end)
  (value alien-field-type start end)
  ...)

This creates and returns a new alien-structure object with the specified name and length (in bytes). The name may or may not be an alien-structure-type that has already been defined; in any event, the name is simply stored away and has no effect on how this object is filled. Each of the values is evaluated (to produce a Lisp object) and then is packed into the specified place in the new alien structure using the setf-transform of the specified alien field type. For example, to send a variable-length string from Lisp to wherever, something like the following function might be employed:

(defun export-string (s)
  (send-external-message wherever
    (pack-alien-structure 'string-message (+ (length s) 4)
      ((length s) 'unsigned-integer 0 4)
      (s 'string 4 (+ 4 (length s))))))

To receive a string of the same format:

(defun import-string ()
  (let* (((foo (receive-external-message wherever))
    (string-length (alien-field foo 'integer 0 4)))
    (alien-field foo 'string 4 (+ string-length 4))))
Index
Index of Concepts
Index of Variables

A
B
C

D
*debug-ignored-functions* 14
*debug-prinlength* 14
*debug-prinlevel* 14

E
*error-cleanup-forms* 8

F
G
H
I
J
K
L

M
*max-step-indentation* 11
*max-trace-indentation* 9

N
O
P
Q
R

S
*step-prinlength* 11
*step-prinlevel* 11

T
*trace-prinlength* 9
*trace-prinlevel* 9
*traced-function-list* 9

U
V
W
Index of Constants

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
Index of Keywords

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
Index of Functions, Macros, and Special Forms
Table of Contents

1. Introduction
   1.1. Obtaining and Running Spice Lisp

2. Implementation Dependent Design Choices
   2.1. Numbers
   2.2. Characters
   2.3. Vector Initialization
   2.4. Packages
   2.5. The Editor
   2.6. Time Functions
   2.7. System Dependent Constants

3. Debugging Tools
   3.1. The Break Loop
      3.1.1. Cleaning Up
   3.2. Function Tracing
      3.2.1. Encapsulation Functions
   3.3. Single Stepper
   3.4. The Debugger
      3.4.1. Movement Commands
      3.4.2. Inspection Commands
      3.4.3. Other Commands
   3.5. Random Features

4. The Compiler
   4.1. Calling the Compiler
   4.2. Open and Closed Coding
   4.3. Compiler Switches
   4.4. Declare switches

5. Efficiency
   5.1. Compile Your Code
   5.2. Avoid Unnecessary Consing
   5.3. Do, Don't Map
   5.4. Think Before You Use a List
      5.4.1. Use Vectors
      5.4.2. Use Structures
      5.4.3. Use Hashtables
      5.4.4. Use Bit-Vectors
   5.5. Simple Vs Complex Arrays
   5.6. To Call or Not To Call
   5.7. Keywords and the Rest
List of Tables
Printer Ruby

Spruce version 12.0 -- spooler version 12.0

File: PRVA:<SLISP.SLM>CLMWIZ.MSS.3

Creation date: 23-Nov-83 12:36 (Wed p.m.)

Printing date: 23-Nov-83 13:00:46 EST

For: STEELE on CMU-CS-C

29 total sheets = 19 pages, 1 copy.
Index

* function 162 * variable 266 ** variable 266
*** variable 266 + function 162
+ variable 266 ++ variable 266
+++ variable 266 - function 162
- variable 266 / function 162
/ variable 267 // variable 267
/// variable 267 /= function 160
1+ function 163 1- function 163
< function 160; 190 <= function 160
= function 160 Compatibility note 11, 21,
36, 37, 43, 55, 61, 66, 86, 89, 97, 102, 106, 107,
112, 115, 116, 126, 127, 131, 134, 137, 150,
151, 161, 162, 168, 169, 174, 181, 182, 184,
201, 202, 204, 214, 215, 220, 222, 226, 230,
235, 237, 251, 252, 278, 282, 286, 293, 297.
301, 302, 311, 317, 319, 320, 327, 343, 344,
345, 347, 348, 350, 351, 358, 360, 361
Implementation note 12, 13, 15, 17, 18, 22,
31, 45, 48, 64, 100, 116, 137, 148, 163, 164,
166, 167, 168, 169, 177, 183, 184, 187, 188,
213, 240, 245, 299, 323, 341, 342, 344, 348,
351, 352, 356 Rationale 23, 27, 28, 51, 81, 85,
93, 100, 157, 161, 163, 176, 185, 308, 316,
347, 354, 361, 362 ^ (new line) format
directive 321 ^& (fresh line) format
directive 321 ^ (case conversion) format
directive 323 ^# (ignore argument) format
directive 322 ^< (justification) format
directive 325 ^<newline> (ignore whitespace) format
directive 321 ^? (indirection) format
directive 322 ^^ (Tilde) format
directive 321 ~A (Aseil) format
directive 314 ~B (Binary) format
directive 315 ~C (Character) format
directive 316 ~D (Decimal) format
directive 315 ~E (Exponential floating-point) format
directive 318 ~F (Fixed-format floating-point) format
directive 316 ~G (Dollars) format
directive 320 ~G (General floating-point) format
directive 319 ~O (Octal) format
directive 315 ~P (Plural) format
directive 316 ~R (Radix) format
directive 315 ~S (S-expression) format
directive 315 ~T (Tabulate) format
directive 321 ~X (Hexadecimal) format
directive 315 ~[ (conditional) format
directive 323 ~^ (loop escape) format
directive 326 ~( (iteration) format
directive 324 ~| (new page) format
directive 321 " macro character 286 #
macro character 288 ' macro character 285
( macro character 285 ) macro
caracter 285 , macro character 288 ;
macro character 285 ' macro character 286
ADA 11, 68 APL 22, 169, 202 ALGOL 30, 47,
107, 174 C language 17, 294 FORTRAN 2, 11,
16, 107, 119, 168, 174, 317, 318, 319, 320
INTERLISP 1, 2, 11, 36, 37, 89, 107, 131, 134,
168, 174, 201, 202, 214, 215, 230, 282, 297,
358 Keywords for defstruct
slot-descriptions 253 LISP 1.5 106, 201 Lisp
Machine LISP 1, 2, 11, 21, 66, 86, 97, 100,
107, 112, 134, 137, 139, 147, 162, 163, 168,
174, 201, 202, 214, 230, 235, 237, 251, 252,
293, 297, 317, 319, 320, 322, 343, 344, 345,
347, 348, 350, 351, 358, 360, 361 MACLISP 1,
bit-nor function 239
bit-not function 240
bit-orc1 function 239
bit-orc2 function 239
bit-vector predicate 62
bit-vector-p function 62
bit-xor function 239 block special
form 98; 32, 47, 55, 71, 99, 102, 103, 107, 108,
109, 113, 114 boole function 179
boole-1 constant 179
boole-2 constant 179
boole-and constant 179
boole-andc1 constant 179
boole-andc2 constant 179
boole-c1 constant 179
boole-c2 constant 179
boole-clr constant 179
boole-eqv constant 179
boole-ior constant 179
boole-nand constant 179
boole-nor constant 179
boole-orc1 constant 179
boole-orc2 constant 179
boole-set constant 179
boole-xor constant 179
both-case-p function 189

boundp function 75; 74
break function 350
*break-on-warnings* variable 350
butlast function 219 byte 181
byte function 181 byte specifiers 181
byte-position function 181
byte-size function 181
caaar function 212; 78
caadar function 212; 78
caaddr function 212; 78
caaadr function 212; 78
caar function 212; 78
cadaar function 212; 78
cadaddr function 212; 78
cadar function 212; 78
cadadar function 212; 78
caddar function 212; 78
caddadr function 212; 78
caddr function 212; 78
cadr function 212; 78
call-arguments-limit constant 90;
s4, 111 car 21, 211 car function 211; 77,
78, 214 case function 98 :
case keyword for write 311
for write-to-string 312
case macro 96; 97, 98, 113, 352, 353, 354
catch 114 catch special form 114; 31, 47,
71, 113 case macro 353; 82, 97, 113
cdaaar function 212; 78
cdaadr function 212; 78
cdaar function 212; 78
cdadar function 212; 78
cdadadr function 212; 79
cdadr function 212; 78
cdar function 212; 78
cddaar function 212; 78
cddadr function 212; 78
cddar function 212; 78
cddadar function 212; 78
cddaddr function 212; 78
cddadr function 212; 79

cddr function 212; 78
cdr 21, 211
cdr function 211; 78, 219
ceiling function 173; 163
cerror function 348; 4, 350
char function 243; 79, 239
char-bit function 195; 79
char-bits function 192; 188
char-bits-limit constant 188; 18, 192
cchar-code function 192; 43, 187
cchar-code-limit constant 187; 192
cchar-control-bit constant 195
cchar-downcase function 193; 189, 246
cchar-equal function 191; 67, 244
cchar-font function 192; 187, 290
char-font-limit constant 187; 18, 192
cchar-greatereqp function 191
cchar-hyper-bit constant 195
cchar-int function 194; 43, 190
cchar-lessp function 191; 210, 245
cchar-meta-bit constant 195
cchar-name function 194
cchar-not-equal function 191
cchar-not-greatereqp function 191
cchar-not-lessp function 191
cchar-super-bit constant 195
cchar-upcase function 193; 189, 246
char/= function 190
char< function 190; 17, 245
char<= function 190
char= function 190; 309
char> function 190 char>= function 190
character coercion to string 247
code 62 character function 192; 42
characterp function 62; 188
check-type macro 351

circle keyword for write 311

for write-to-string 312
cis function 167 cleanup handler 115
clear-input function 309
clear-output function 312
close function 273; 339, 341, 342
clrhash function 231
code-char function 192
coerce function 42; 43, 172, 192, 200, 201, 209, 247 comments 285 common data
type predicate 63
commonp function 63
compile function 355
compile-file function 356 compiled
function predicate 63
compiled-function-p function 63
compiler-let special form 92; 47
complex function 176; 16, 39 complex
number predicate 62
complexp function 62; 159

concat-namekeyword for defstruct 253
concatenate function 200; 216
cond macro 95; 59, 68, 69, 97, 101, 113
conditional and 68 or 68
during read 293
conjugate function 163 cons 21, 211
predicate 61 cons function 213; 39
consp function 61
constantp function 265; 138 constructor
function 250
constructokeyword for defstruct 254;
252, 257 control structure 71 copier
function 250
copier keyword for defstruct 254
copy-alist function 216
copy-list function 216
directory-name string function 337
disassemble function 356 displaced
array 234

displaced-index-adjust-array 241
for make-array 234

for displaced-keyword adjust-array 241
for make-array 234 do macro 100;
32, 71, 76, 99, 107, 113, 125 do* macro 100;
99, 125 do-all-symbols macro 152; 105,
125,

do-external-symbols macro 152; 105,
125 do-symbols macro 152; 105, 125
documentation function 356; 41, 55, 57,
79, 118, 251 dolist macro 104; 99, 113,
119, 125 dotimes macro 104; 99, 113, 125
dotted list 211
double-float-epsilon constant 186
double-float-negative-equivalent 186
dpb function 182; 79, 182
dribble function 359 dynamic exit 114
dcase macro 353; 97, 113
d function 358 eighth function 214; 78

:element-type keyword &adjust-array 241
for make-array 233
for open 339 elt function 199; 79,
214, 236, 244 empty list predicate 61
encode-universal-time function 361
:end keyword for count 207
for count-if 207
for count-if-not 207
for delete 204
for delete-duplicates 205
for delete-if 204
for delete-if-not 204
for fill 203 for find 206
for find-if 206
for find-if-not 206
for nstring-capitalize 247
for nstring-downcase 247
for nstring-upcase 247
for nsubstitute 206
for nsubstitute-if 206
for nsubstitute-if-not 206
for parse-integer 310
for position 206
for position-if 206
for position-if-not 206
for read-from-string 309
for reduce 202 for remove 203
for remove-duplicates 205
for remove-if 203
for remove-if-not 203
for string-capitalize 246
for string-downcase 246
for string-upcase 246
for substitute 205
for substitute-if 205
for substitute-if-not 205
for write-line 312
for write-string 312
for with-input-from-string 272
:end keyword for mismatch 207
for replace 203 for search 207
for string-equal 244
for string-greaterp 245
for string-lessp 245
for string-not-equal 245
for string-not-greaterp 245
for string-not-lessp 245
for string/= 245
define keyword for mismatch 207
for replace 203 for search 207
for string-equal 244
for string-greaterp 245
for string-lessp 245
for string-not-equal 245
for string-not-greaterp 245
for string-not-lessp 245
for string/= 245
for string< 245
for string<= 245
for string= 244
for string> 245
for string=> 245
end2 keyword for mismatch 207
for replace 203 for search 207
for string-equal 244
for string-greaterp 245
for string-lessp 245
for string-not-equal 245
for string-not-greaterp 245
for string-not-lessp 245
for string/= 245
for string< 245
for string<= 245
for string= 244
for string> 245
for string=> 245
enough-namestring function 337
environment structure 71 eq function 63; 65
compared to equal 63
eql function 65; 36, 157, 161, 187, 191
equal function 66; 191, 213, 244, 275, 334
equalp function 67; 223
error function 348; 4 :
errorkeyword for if-does-not-exist
option to open 341 for if-exists
option to open 340
*error-output* variable 270; 350 :
escape keyword for write 311
for write-to-string 312
etypecase macro 353; 98, 113
eval function 263; 112, 117
eval-when special form 57; 47, 113, 119,
127, 148, 292, 355 evalhook function 264;
123, 264 *evalhook* variable 263
evenp function 159 every function 201;
68 exp function 165
export function 151; 144, 145
expt function 165; 158 extent 29
false when a predicate is 59
fboundp function 75
fceiling function 175
*features* variable 363; 293
ffloor function 175
fifth function 214; 78
file-author function 344
file-length function 344
file-namestring function 337
file-position function 344; 340
file-write-date function 344
fill function 203 fill pointer 240
fill-pointer function 241; 79, 243 :
fill-point keyword adjust-array 241
for make-array 234
find function 206; 222, 225, 226
find-all-symbols function 152
find-if function 206
find-if-not function 206
find-package function 149; 141
find-symbol function 150
finish-output function 312
first function 214; 78, 211 fixnum 11
flet special form 93; 47, 49, 75, 117, 125,
129 float function 172; 168
float-digits function 175
float-precision function 175
float-radix function 175; 13
float-sign function 175; 159, 298
floating-point number 13 predicate 62
floate function 62; 159
floor function 173; 43, 110, 163, 174 flow of control 71 fmakunbound function 77;
75 force-output function 312
format function 313; 247, 313, 328, 347,
349, 350, 351 formatted output 313
fourth function 214; 78
fresh-line function 312; 321, 327, 328:
from-end keyword for count 207
for count-if 207
for count-if-not 207
for delete 204
for delete-duplicates 205
for delete-if 204
for delete-if-not 204
for find 206 for find-if 206
for find-if-not 206
for mismatch 207
for nsubstitute 206
for nsubstitute-if 206
for nsubstitute-if-not 206
for position 206
for position-if 206
for position-if-not 206
for reduce 202 for remove 203
for remove-duplicates 205
for remove-if 203
for remove-if-not 203
for search 207
for substitute 205
for substitute-if 205
for substitute-if-not 205
fraction function 175
ftruncate function 175
funcall function 89; 26, 49, 60, 112, 117,
123, 264 function predicate 63 function
declaration 129 function function 26
function special form 72; 33, 47, 49, 52
function type declaration 129
functionp function 63; 49
gcd function 164 general array 233
gensym function 138; 85, 86, 120, 138 :
gensym keyword for write 311
for write-to-string 312
gensemp function 138; 85, 86, 138
get function 134; 78, 79, 134, 135
get-decoded-time function 361
get-dispatch-macro-char function 297
get-internal-real-time function 362
get-internal-run-time function 361
get-macro-character function 295
get-output-stream-string function 272
get-properties function 136
get-setf-method function 88
get-setf-method-multiple function 88
get-universal-time function 361
getf function 135; 79, 82, 134, 135, 136
gethash function 231; 79 go special
graphic-char-p function 188; 190, 194
hash table 229, 232 predicate 230
hash-table-count function 232
hash-table-p function 230; 63 home
directory 337 host (pathname
component) 332
host keyword for make-pathname 336
host-namestring function 337
identity function 363 if special form 95; 47, 59, 68, 69, 95, 113, 121
if-does-not-exist keyword fnread 345
for open 341
if-exists keyword for open 340
ignore declaration 130
imagpart function 177 implicit progn 71, 91, 92, 93, 95, 96, 101
import function 151; 142, 143, 145
in-package function 149
incf macro 163; 82, 84
include keyword for defstruct 254; 28, 260
indexofpath-input-from-string 272
index offset 235 indicator 133 indirect array 234
init-file-pathname function 363
initial-context keyword just-array 241
for make-array 234; 300
initial-element keyword just-array 241
for make-list 216
for make-sequence 200
for make-string 245
for make-array 234
initial-offskew keyword fnstruct 257;
260
initial-value keyword forreduce 202
inline declaration 129
input keyword for direction option to open 339
input-stream-p function 273
inspect function 358
int-char function 194 integer 21
predicate 61
integer-decode-float function 175
integer-length function 181; 184
integerp function 61; 159
intern function 150; 63, 137, 138, 140, 145
internal-time-units-per-second 261;
359, 362
intersection function 223
io keyword for direction option to open 339 isqrt function 166
iteration 99
junk-allowed keyword parse-integer 310
key keyword for adjoin 222
for count 207 for count-if 207
for count-if-not 207
for delete 204
for delete-duplicates 205
for delete-if 204
for delete-if-not 204
for find 206 for find-if 206
for find-if-not 206
for intersection 223
for member 222
for member-if 222
for member-if-not 222
for merge 209 for mismatch 207
for nintersection 223
for nset-difference 224
for nset-exclusive-or 224
for nsublis 222 for nsubst 221
for nsubst-if 221
for nsubst-if-not 221
for nsubstitute 206
for nsubstitute-if 206
for nsubstitute-if-not 206
for union 223
for position 206
for position-if 206
for  position-if-not 206
for  remove 203
for  remove-duplicates 205
for  remove-if 203
for  remove-if-not 203
for  search 207
for  set-difference 224
for  set-exclusive-or 224
for  sort 208
for  stable-sort 208
for  sublis 221  for-subsetp 225
for  subst 220  for-subst-if 220
for  subst-if-not 220
for  substitute 205
for  substitute-if 205
for  substitute-if-not 205
for  union 223
keywodp function 138  labels special
form 93; 47, 49, 75, 117, 125, 129
lambda-list-keywords constant 54;
119
lambda-parameters-limit constant 54;
90, 111  last  function 215
1cm function 164  ldb function 182; 79, 87
1db-test  function 182
1diff  function 219; 222
least-negative-double-constant 186
least-negative-long-float-constant 186
least-negative-short-float-constant 185
least-negative-single-float-constant 186
least-positive-double-constant 186
least-positive-long-float-constant 186
least-positive-short-float-constant 185
least-positive-single-float-constant 186
length  function 200; 199, 213, 214, 237:
length keyword for write 311
for write-to-string 312
let special form 91; 31, 46, 47, 92, 93, 99,
103, 107, 108, 109, 113, 125
let* function 86  let* special form 92;
47, 52, 109, 113, 125 :
level keyword for write 311
for write-to-string 312
lisp-implementation-type function 362
lisp-implementation-version function 362
list 21, 211  predicate 61  See
also: dotted list  list  function 215  list
syntax 285  list* function 215; 89
list-all-packages function 150
list-length  function 213
listen function 309  listp function 61;
211  load  function 345; 148, 345
*load-pathname-defaults* variable 338
*load-verbose* variable 345
locally macro 127; 125
log function 165; 158
logand function 178; 240
logandc1 function 178
logandc2 function 178
logbitp function 180
logcount function 180
logeqv function 178  logical
operators on nil and non-nil
values 67  logior function 177
lognand function 178
lognor function 178
lognot function 180; 240
logorc1 function 178
logorc2 function 178
logtest function 180
INDEX

logxor function 177
long-float-epsilon constant 186
long-float-negative-epsilon constant 186
long-site-name function 363
loop macro 100; 99, 101, 103
lower-case-p function 189; 190, 193, 297 machine-instance function 362
machine-type function 362
machine-version function 362 macro character 284
macro-function function 118; 47, 75
macroexpand function 123; 48, 118, 264
macroexpand-1 function 123
*macroexpand-hook* variable 123
macrolet special form 93; 47, 117, 118, 119, 123, 125 make-array function 233; 37, 38, 54, 241, 245, 292
make-broadcast-stream function 271
make-char function 193
make-concatenated-stream function 271
make-dispatch-macro-character 296; 297 make-echo-stream function 271
make-hash-table function 230
make-list function 216
make-package function 149
makepathname function 336
make-random-state function 184; 301
make-sequence function 200
make-string function 245
make-string-input-stream function 271
make-string-output-stream function 271
make-symbol function 137
make-synonym-stream function 271; 270
make-two-way-stream function 271
makunbound function 77; 46, 74, 75, 93
map function 201; 43, 105, 117, 264
mapc function 105; 201
mapcan function 105
mapcar function 105
mapcon function 105
maphash function 231
mapl function 105; 201
maplist function 105 mapping 105
mask-field function 182; 79
max function 161 member function 222; 59, 225 member-if function 222
member-if-not function 222
merge function 209
merge-pathnames function 336; 343
merging of pathnames 332 sorted sequences 209 min function 161
minusp function 159
mismatch function 207; 223
mod function 174
*modules* variable 153
most-negative-double-float constant 186
most-negative-fixnum constant 185; 11, 40
most-negative-long-float constant 186
most-negative-short-float constant 185
most-negative-single-float constant 186
most-positive-double-float constant 186
most-positive-fixnum constant 185; 11, 40, 57
most-positive-long-float constant 186
most-positive-short-float constant 185
most-positive-single-float constant 186
multiple values 110
multiple-value-bind macro 112; 110,
pathname-type function 336
pathname-version function 336
pathnamep function 336; 63
peek-char function 308
phase function 166 pi constant 168; 32, 265 plist 133 plusp function 159
pop macro 218; 82 position of a byte 181 position function 206; 37, 222, 226 position-if function 206
position-if-not function 206
pprint function 311 predicate 59 : predicate-keyword fordefstruct 254
predicates true and false 59 :
preserve-whiteom-string 309 : pretty keyword for write 311
for write-to-string 312
print function 311; 12, 301, 312, 315, 317, 318 print-to-string function 312; 247.
princ function 311; 301, 312, 314
princ-to-string function 312; 247
print function 311; 185, 269, 275 :
print keyword for load 345 print name 133, 136, 243 coercion to string 247 *print-array* variable 304; 299, 300, 311 *print-base* variable 302; 298, 299, 311 *print-case* variable 302;
299, 311 *print-circle* variable 302; 217, 299, 311
*print-escape* variable 301; 256, 298, 299, 311
: print-function-keyword fordefstruct 256;
26 *print-gensym* variable 303; 299, 311 *print-length* variable 303; 281, 293, 300, 311
*print-level* variable 303; 256, 294, 300, 311 *print-pretty* variable 302; 256, 311 *print-radix* variable 302;
298, 311 printed representation 275
printer 275, 298 :
probe keyword for direction option to open 339
probe-file function 343
proclaim function 127; 50, 56
proclamation 127 prog macro 108; 32, 99, 113, 125 prog* macro 108; 113, 125
prog1 macro 90; 71, 112, 113, 114
prog2 macro 91; 71, 114 progn special form 90; 47, 55, 71, 98, 99, 101, 113
progv special form 93; 47, 77, 113
property 133 property list 133 compared to association list 133 compared to hash table 229 provide function 153
psatf macro 80; 76 psetq macro 76; 101, 103 push macro 217; 82
pushnew macro 218; 222
*query-io* variable 270, 327, 328
querying the user 327 quote character 285
quote special form 72; 47, 64, 75 :
radius-keyword forparse-integer 310
for write 311
for write-to-string 312
random function 183
random-state predicate 185
*random-state* variable 184
random-state-p function 185; 63
rank 22 rassoc function 227; 225
rassoc-if function 227
rassoc-if-not function 227 ratio 12
rational 12 predicate 62
rational function 172; 43
rationalize function 172
rationalp function 62; 159
read function 305; 6, 10, 23, 57, 72, 74, 136,
137, 185, 269, 270, 276, 282, 285, 301, 302,
304, 311, 315 *read-base* variable 282;
280, 282, 299 read-byte function 310;
339, 340 read-char function 308; 269,
270, 309, 339, 344
read-char-no-hang function 309
*read-default-float-convertible 305;
14, 298, 318
read-delimited-list function 307;
295 read-from-string function 309
read-line function 308; 304, 312 :
read-only keyword for defstruct
slot-descriptions 253
read-preserving-whitespace function 306;
277, 309 *read-suppress* variable 282;
293, 305 reader 275, 276 readable 294
predicate 295
*readtable* variable 294; 295
readtablep function 295; 63
realpart function 177 record
structure 249 reduce function 202; 264 :
rehash-keyword-hashing-table 230
rehash-threshold function 230
rehash-threshold function 230
rem function 174 remf macro 136; 82,
134 remhash function 231
remove function 203; 198
remove-duplicates function 205
remove-if function 203
remove-if-not function 203; 106
remprop function 135; 136 :
rename keyword for if-exists
option to open 340 :
rename-and-delektoword if-exists
option to open 340
rename-file function 343
rename-package function 149; 141
replace function 203; 79, 199
require function 153
rest function 215; 78, 211
return macro 99; 48, 71, 100, 101, 102,
103, 108, 113, 114, 152
return-from special form 99; 4, 32, 47,
55, 71, 99, 101, 110, 113, 115
revappend function 217
reverse function 200 return function 38
rotate macro 82 round function 173;
162, 163 rplaca function 220; 77, 85, 211
rplacd function 220; 211
sample-constant constant 4
sample-function function 4
sample-macro macro 4
sample-special-form special form 4
*sample-variable* variable 4
sbit function 239; 79, 238
scale-float function 175
schar function 243; 79, 238, 239
scheme1 scope 29 search function 207;
199 second function 214; 78 set list
representation 222 set function 76; 74, 75,
77 set-char-bit function 195; 79, 195
set-difference function 224
set-dispatch-macro-character function 297
set-exclusive-or function 224
set-macro-character function 295;
58, 276, 297
set-syntax-from-char function 295;
for substitute-if 205
for substitute-if-not 205
for write-line 312
for write-string 312
for with-input-from-string 272

start1 keyword for mismatch 207
for replace 203 for search 207
for string-equal 244
for string-greaterp 245
for string-lessp 245
for string-not-equal 245
for string-not-greaterp 245
for string-not-lessp 245
for string/= 245
for string< 245
for string<= 245
for string= 244
for string> 245
for string>= 245 :

start2 keyword for mismatch 207
for replace 203 for search 207
for string-equal 244
for string-greaterp 245
for string-lessp 245
for string-not-equal 245
for string-not-greaterp 245
for string-not-lessp 245
for string/= 245
for string< 245
for string<= 245
for string= 244
for string> 245
for string>= 245 step macro 357;

stream keyword for write 311
stream-element-type function 273;
340 stream function 273; 63 string, 243
predicate 62 string function 247; 243
string syntax 286
string-capitalise function 246; 247,
302, 323 string-char-p function 188;
63, 243 string-downcase function 246
string-equal function 244; 65
string-greaterp function 245
string-left-trim function 246
string-lessp function 245
string-not-equal function 245
string-not-greaterp function 245
string-not-lessp function 245
string-right-trim function 246
string-trim function 246
string-upcase function 246
string/= function 245
string< function 245
string<= function 245, 149
string= function 244; 65, 149
string> function 245
string>= function 245
stringp function 62; 243 structure 249
structured pathname components 333
sublis function 221
subseq function 199; 79
subsetp function 225
subst function 220; 221
subst-if function 220
subst-if-not function 220
substitute function 205; 221
substitute-if function 205
substitute-if-not function 205
substitution function 220

: supersede keyword for if-exists option to open 341
svref function 238;
79 sxhash function 232 symbol 9, 133
 coercion to a string 243 coercion to
string 247 predicate 61
symbol-function function 75; 26, 72,
75, 79, 133 symbol-name function 136
symbol-package function 138; 142
symbol-plist function 135; 79
symbol-value function 74; 77, 79, 133,
263 symbolp function 61 t constant 60;
57, 265 tagbody special form 107; 32, 47,
99, 100, 101, 103, 107, 108, 109
tailp function 222 tan function 167
tanh function 169 tenth function 214;
78 *terminal-io* variable 270; 304,
310, 327 terpri function 312; 311, 321:
test keyword for adjoin 222
for assoc 226 for count 207
for delete 204
for delete-duplicates 205
for find 206
for intersection 223
for make-hash-table 230
for member 222
for mismatch 207
for nintersection 223
for nset-difference 224
for nset-exclusive-or 224
for nsbunis 222 for nsubst 221
for nsubstitute 206
for nunion 223
for position 206
for rassoc 227 for remove 203
for remove-duplicates 205
for search 207
for set-difference 224
for set-exclusive-or 224
for sublis 221 for subsetp 225
for subst 220
for substitute 205
for tree-equal 213
for union 223 : test-not keyword for adjoin 222
for assoc 226 for count 207
for delete 204
for delete-duplicates 205
for find 206
for intersection 223
for member 222
for mismatch 207
for nintersection 223
for nset-difference 224
for nset-exclusive-or 224
for nsbunis 222 for nsubst 221
for nsubstitute 206
for nunion 223
for position 206
for rassoc 227 for remove 203
for remove-duplicates 205
for search 207
for set-difference 224
for set-exclusive-or 224
for sublis 221 for subsetp 225
for subst 220
for substitute 205
for tree-equal 213
for union 223 the special form 131;
vector-push function 241; 273, 313
vector-push-extend function 241
vectorp function 62 :
verbose keyword for load 345
version (pathname component) 332 :
version keyword formake-pathname 336
warn function 350 when macro 95; 99, 95,
95, 113, 350
with-input-from-string macro 272
with-open-file macro 342; 30, 271, 342
with-open-stream macro 272
with-output-to-string macro 272
write function 311; 312
write-byte function 313; 339, 340
write-char function 312; 269, 308, 339,
344 write-line function 312; 308
write-string function 312
write-to-string function 312
y-or-n-p function 327 yes-or-no
functions 327 yes-or-no-p function 328;
270 zerop function 159
Table of Contents