

LISP RELEASE NOTES



Address comments to:
ENVOS
User Documentation
1157 San Antonio Rd.
Mountain View, CA 94043
415-966-6200

LISP RELEASE NOTES

Medley Release 1.0

400006

September 1988

Copyright © 1988 by ENVOS Corporation.

All rights reserved.

Envos is a trademark of Envos Corporation.

Medley is a trademark of Envos Corporation.

Xerox® is a registered trademark of Xerox Corporation.

Sun® is a registered trademark of Sun Microsystems Inc.

DEC®, VAX®, VMS®, and VT100® are registered trademarks of Digital Equipment Corporation.

UNIX® is a registered trademark of AT&T Bell Laboratories.

Copyright protection includes material generated from the software programs displayed on the screen, such as icons, screen display looks, and the like.

The information in this document is subject to change without notice and should not be construed as a commitment by Envos Corporation. While every effort has been made to ensure the accuracy of this document, Envos Corporation assumes no responsibility for any errors that may appear.

Text was written and produced with Envos formatting tools using Xerox printers to produce text masters. The typeface is Modern.

Preface	xvii
How the Release Notes are Organized	xvii
Notational Conventions	xviii
How to Use the Release Notes	xviii
Related Literature	xix
1. Introduction	1-1
Summary of Medley Changes	1-1
2. Notes and Cautions	2-1
Changes and Cautions in the Medley Release	2-1
Changes and Cautions in the Lyric Release	2-1
3. Common Lisp/Interlisp-D Integration	3-1
Chapter 2 Litatoms	3-1
Section 2.1 Using Litatoms as Variables	3-2
Section 2.3 Property Lists	3-2
Section 2.4 Print Names	3-2
Section 2.5 Characters	3-3
Chapter 4 Strings	3-3
Chapter 5 Arrays	3-3
Chapter 6 Hash Arrays	3-4
Chapter 7 Numbers and Arithmetic Functions	3-4
Section 7.2 Integer Arithmetic	3-4
Chapter 10 Function Definition, Manipulation, and Evaluation	3-5
Section 10.1 Function Types	3-5
Section 10.6 Macros	3-5
Section 10.6.1 DEFMACRO	3-5
Chapter 11 Stack Functions	3-5
Section 11.1 The Spaghetti Stack	3-5
Chapter 12 Miscellaneous	3-6
Section 12.4 System Version Information	3-6
Section 12.8 Pattern Matching	3-6
Chapter 13 Interlisp Executive	3-7
Chapter 14 Errors and Breaks	3-9
Section 14.3 Break Commands	3-9
Section 14.6 Creating Breaks with BREAK1	3-9
Section 14.7 Signalling Errors	3-9
Section 14.8 Catching Errors	3-10

Section 14.9 Changing and Restoring System State	3-11
Section 14.10 Error List	3-11
Chapter 15 Breaking Functions and Debugging	3-13
Section 15.1 Breaking Functions and Debugging	3-13
Section 15.2 Advising	3-14
Chapter 16 List Structure Editor	3-15
Switching Between Editors	3-16
Packages	3-16
Starting a Lisp Editor	3-16
Mapping the Old Edit Interface to ED	3-18
Editing Values Directly	3-18
Section 16.18 Editor Functions	3-19
Chapter 17 File Package	3-19
Reader Environments and the File Manager	3-20
Modifying Standard Readtables	3-22
Programmer's Interface to Reader Environments	3-23
Section 17.1 Loading Files	3-24
Integration of Interlisp and Common Lisp LOAD Functions	3-24
Section 17.2 Storing Files	3-25
Section 17.8.2 Defining New File Manager Types	3-26
Definers: A New Facility for Extending the File Manager	3-26
Chapter 18 Compiler	3-31
Warning when Loading Compiled Files	3-32
Warning with Declarations	3-32
Section 18.3 Local Variables and Special Variables	3-33
Chapter 19 Masterscope	3-33
Chapter 21 CLISP	3-33
Chapter 22 Performance Issues	3-36
Section 22.3 Performance Measuring	3-36
Chapter 24 Streams and Files	3-37
Section 24.15 Deleting, Copying, and Renaming Files	3-38
Chapter 25 Input/Output Functions	3-38
Variables Affecting Input/Output	3-38
Integration of Common Lisp and Interlisp Input/Output Functions	3-40
Section 25.2 Input Functions	3-40
Section 25.3 Output Functions	3-41
Printing Differences Between IL:PRIN2 and CL:PRIN1	3-42
Internal Printing Functions	3-42
Printing Differences Between Koto and Lyric	3-42
Bitmap Syntax	3-43

Section 25.8 Readtables	3-43
Differences Between Interlisp and Common Lisp Readtables	3-44
Section 25.8.2 New Readtable Syntax Classes	3-45
Additional Readtable Properties	3-45
Section 25.8 Predefined Readtables	3-47
Koto Compatibility Considerations	3-48
Specifying Readtables and Packages	3-48
The T Readtable	3-48
PQUOTE Printed Files	3-49
Back-Quote Facility	3-49
Chapter 28 Windows and Menus	3-49
Section 28.5.1 Menu Fields	3-49
4. Changes to Interlisp-D in Lyric/Medley	4-1
Chapter 3 Lists	4-1
Section 3.2 Building Lists From Left To Right	4-1
Section 3.10 Sorting Lists	4-1
Chapter 6 Hash Arrays	4-1
Section 6.1 Hash Overflow	4-2
Chapter 7 Integer Arithmetic	4-2
Section 7.3 Logical Arithmetic Functions	4-3
Section 7.5 Other Arithmetic Functions	4-3
Chapter 8 Record Package	4-3
Chapter 9 Conditionals and Iterative Statements	4-3
Section 9.2 Equality Predicates	4-3
Section 9.8.3 Condition I.s. oprs	4-3
Chapter 10 Function Definition, Manipulation, and Evaluation	4-4
Section 10.2 Defining Functions	4-4
Section 10.5 Functional Arguments	4-4
Section 10.6.2 Interpreting Macros	4-4
Chapter 11 Variable Bindings and the Interlisp Stack	4-4
Section 11.2.1 Searching the Stack	4-5
Section 11.2.2 Variable Bindings in Stack Frames	4-5
Section 11.2.5 Releasing and Reusing Stack Pointers	4-5
Section 11.2.7 Other Stack Functions	4-5
Chapter 12 Miscellaneous	4-6
Section 12.2 Idle Mode	4-6
Section 12.3 Saving Virtual Memory State	4-7
Section 12.4 System Version Information	4-7
Chapter 13 Interlisp Executive	4-8
Chapter 14 Errors and Breaks	4-8

Section 14.5 Break Window Variables	4-8
Section 14.8 Catching Errors	4-8
Chapter 17 File Package	4-9
Section 17.8.1 Functions for Manipulating Typed Definitions	4-9
Section 17.8.2 Defining New File Package Types	4-9
Section 17.9.2 Variables	4-9
Section 17.9.8 Defining New File Package Commands	4-9
Section 17.11 Symbolic File Format	4-9
Section 17.11.3 File Maps	4-10
Chapter 18 Compiler	4-10
Chapter 21 CLISP	4-10
Section 21.8 Miscellaneous Functions and Variables	4-10
Chapter 22 Performance Issues	4-11
Section 22.1 Storage Allocation and Garbage Collection	4-11
Section 22.5 Using Data Types Instead of Records	4-11
Chapter 23 Processes	4-12
Section 23.6 Typein and the TTY Process	4-12
Section 23.8 Process Status Window	4-12
Chapter 24 Streams and Files	4-13
Section 24.7 File Attributes	4-13
Section 24.9 Local Hard Disk Device	4-13
Section 24.10 Floppy Disk Device	4-13
Section 24.12 Temporary Files and CORE Device	4-13
Section 24.18.1 Pup File Server Protocols	4-14
Section 24.18.1-2 Use of BREAKCONNECTION with File Servers	4-14
Section 24.18.2 NS File Server Protocols	4-15
Section 24.18.3 Operating System Designations	4-15
Chapter 25 Input/Output Functions	4-15
Section 25.2 Input Functions	4-15
Section 25.3.2 Printing Numbers	4-15
Section 25.3.4 Printing Unusual Data Structures	4-15
Section 25.4 Random Access File Operations	4-16
Section 25.6 PRINTOUT	4-16
Section 25.8.3 READ Macros	4-16
Chapter 26 User Input/Output Packages	4-16
Section 26.3 ASKUSER	4-16
Section 26.4 TTYIN Display Typein Editor	4-16
Section 26.4.3 Display Editing Commands	4-17
Section 26.4.5 Useful Macros	4-18
Chapter 27 Graphic Output Operations	4-18

Section 27.1.3 Bitmaps	4-18
Section 27.3 Accessing Image Stream Fields	4-18
Section 27.6 Drawing Lines	4-19
Section 27.7 Drawing Curves	4-19
Section 27.8 Miscellaneous Drawing and Printing Operations	4-19
Section 27.12 Fonts	4-21
Section 27.13 Font Files and Font Directories	4-23
Section 27.14 Font Classes	4-23
Section 27.14 Font Profiles	4-23
Chapter 28 Windows and Menus	4-24
Section 28.4 Windows	4-24
Section 28.4.5 Reshaping Windows	4-24
Section 28.4.8 Shrinking Windows Into Icons	4-24
Section 28.4.11 Terminal I/O and Page Holding	4-25
Section 28.5 Menus	4-26
Section 28.6.2 Attached Prompt Windows	4-28
Section 28.6.3 Window Operations and Attached Windows	4-28
Chapter 29 Hardcopy Facilities	4-29
Chapter 30 Terminal Input/Output	4-29
Section 30.1 Interrupt Characters	4-29
Section 30.2.3 Line Buffering	4-30
Section 30.4.1 Changing the Cursor Image	4-30
Section 30.5 Keyboard Interpretation	4-31
Section 30.6 Display Screen	4-31
Section 30.7 Miscellaneous Terminal I/O	4-31
Chapter 31 Ethernet	4-32
Section 31.3.1 Name and Address Conventions	4-32
Section 31.3.2 Clearinghouse Functions	4-33
Section 31.3.3 NS Printing	4-34
Section 31.3.5.3 Performing Courier Transactions	4-34
Section 31.3.5.3.3 Using Bulk Data Transfer	4-34
Section 31.5 Pup Level One Functions	4-34
Section 31.6.1 Creating and Managing XIPs	4-35

5. Library Modules	5-1
Modules That are New, Moved, or Replaced	5-1
Modules Moved From the Library to LispUsers	5-1
Modules Moved From LispUsers to the Library	5-1
Modules Moved to Their Own Manuals	5-1
Modules Moved From the Sysout into the Library	5-1
Modules Moved From the Library into the Sysout	5-2

Modules Replaced	5-2
New Modules	5-2
Details of Change	5-2
4045XLPStream	5-2
Cash-File	5-2
Centronics	5-3
Chat	5-3
CopyFiles	5-3
DataBaseFns	5-3
EditBitMap	5-3
FileBrowser	5-3
FTPServer	5-4
FX-80Driver	5-4
GCHax	5-5
Grapher	5-5
Hash	5-5
Hash-File	5-5
Kermit	5-5
MasterScope	5-5
NSMaintain	5-5
RS232	5-6
Spy	5-6
TableBrowser	5-6
TCP- IP	5-7
TExec	5-8
TextModules	5-8
Virtual Keyboards	5-8
Where-Is	5-8
Additional Notes	5-8
Koto CML Library Module	5-8

6. User's Guides

6-1

A User's Guide to TEdit—Release Notes	6-1
Expanded Characters	6-1
Put Submenu	6-1
Get Submenu	6-2
Clarified Paragraph Looks Menu Options	6-2
New Page: Before After	6-3
Displaymode: Hardcopy	6-3
Clarified Page Layout Menu Options	6-3

Added Items to Programmer's Interface	6-3
Corrected the AFTERQUITFN Property	6-3
Corrected the TEXTOBJ Data Structure	6-4
Corrected the TITLEMENUFN Property	6-4
Expanded the TEDIT.INCLUDE Function	6-4
Expanded the TEDIT.PARALOOKS Function	6-4
Expanded the TEXTPROP Function	6-5
Added Documentation for Global Variables	6-5
Changes to Programmer's Interface to TEdit	6-5
STREAM and TEXTOBJ	6-5
Changes, Additions and Corrections to TEdit Functions	6-5
Changes in Documentation of TEdit Functions	6-7
New Features	6-8
A User's Guide to Sketch—Release Notes	6-10
Manipulating Sketch Elements	6-10
Adding and Deleting Control Points	6-10
Deleting Control Points	6-10
Defaults Command	6-10
Better Feedback for Creating Wires, Circles and Ellipses	6-10
Arrowheads	6-10
Deleting Characters During Type-in	6-10
Using Bit Maps in a Sketch	6-11
Zooming Bitmaps	6-11
Changing Bitmaps	6-11
Freezing Sketch Elements	6-11
Aligning Sketch Elements	6-11
Placing Multiple Copies of Elements	6-11
Making the Window Fit the Sketch	6-12
Overlaying Figure Elements	6-12
Changing How Elements Overlap	6-12
Loading the Sketch Library Module	6-12
The Programmer's Interface	6-13
New Behavior for the Get Command	6-13
Establishing Initial Defaults for Sketch	6-13
1108 User's Guide Release Notes	6-14
What to Look For	6-14
File System	6-14
System Tools	6-14
Input/Output	6-15
Machine Diagnostics	6-15

1186 User's Guide Release Notes	6-16
What to Look For	6-16
File System	6-16
Software Installation	6-16
System Tools	6-17
Input/Output	6-17
Diagnostics	6-17
7. Common Lisp Implementation	7-1
New Features Since Lyric	7-1
Common Lisp Definers	7-1
Compile-Definer	7-2
Compile-Form	7-2
Define-File-Environment	7-2
Site-Name Special Uses	7-3
Record Access	7-3
Define-Record	7-3
Record-Fetch	7-4
Record-FFetch	7-4
Record-Create	7-4
Array Reference	7-4
Shadowing of Global Macros	7-4
Evaluating Load-time Expressions	7-4
Common Lisp Defstruct Options	7-4
Defstruct Options	7-5
Defstruct Slot Options	7-5
Warning When Using Defstruct	7-6
Macros for Collecting Objects	7-6
xcl:with-collection	7-6
Macros for Writing Macros	7-7
xcl:once-only	7-7
Common Lisp Append Datatypes	7-8
Closure Cache	7-8
Symbols and Packages	7-8
Pkg -goto and In -package	7-8
Defpackage Export Argument	7-9
Debugging Tools	7-9
Breaking	7-9
Advising	7-9
Argument Names Displayed for Interpreted Functions	7-10
Lexical Variables Evaluated by Debugger	7-10

Pathname Component Fixed in FS-ERROR	7-10
Compiler Optimizations	7-10
Warning when using LABELS Construct	7-10
COMS added to dfasl files	7-11
Loadflg argument	7-11
Changes in MAP, WRITE-STRING, COERCE, GENSYM, DEFERREDCONSTANT	7-11
Compiler keeps Special &REST arguments	7-12
Compiler ignores TEdit formatting	7-12
Compiler notices Tail-recursive Lexical Functions	7-12
Compiler Error Message	7-12
Format ~C and WRITE-CHAR	7-13
WITH-OUTPUT-TO-STRING / WITH-INPUT-FROM-STRING	7-13

A. The Exec A-1

Input Formats	A-2
Multiple Execs and the Exec's Type	A-4
Event Specification	A-4
Exec Commands	A-5
Variables	A-9
Fonts in the Exec	A-10
Changing the Exec	A-11
Defining New Commands	A-11
Undoing	A-12
Undoing in the Exec	A-13
Undoing in Programs	A-13
Undoable Versions of Common Functions	A-14
Modifying the UNDO Facility	A-14
Undoing Out of Order	A-16
Format and Use of the History List	A-16
Making or Changing an Exec	A-18
Editing Exec Input	A-20
Editing Your Input	A-21
Using the Mouse	A-21
Editing Commands	A-22
Cursor Movement Commands	A-22
Buffer Modification Commands	A-23
Miscellaneous Commands	A-23
Useful Macros	A-24
?= Handler	A-24

Assorted Flags	A-24
B. SEdit—The Lisp Editor	B-1
16.1 SEDIT—The Structure Editor	B-1
16.1.1 An Edit Session	B-1
16.1.2 SEdit Carets	B-2
16.1.3 The Mouse	B-3
16.1.4 Gaps	B-4
16.1.5 Broken Atoms	B-4
16.1.6 Special Characters	B-5
16.1.7 Commands	B-6
16.1.8 Editing Commands	B-7
16.1.9 Completion Commands	B-7
16.1.10 Undo Commands	B-7
16.1.11 Find Commands	B-8
16.1.12 General Commands	B-9
16.1.13 Miscellaneous	B-11
16.1.14 Help Menu	B-11
16.1.15 Command Menu	B-12
16.1.16 SEdit Programmer's Interface	B-12
16.1.17 SEdit Window Region Manager	B-12
16.1.18 Options	B-13
16.1.19 Control Functions	B-14
Warning with Declarations	B-18
C. ICONW	C-1
28.4.16 Creating Icons with ICONW	C-1
28.4.16.1 Creating Icons	C-1
28.4.16.2 Modifying Icons	C-2
28.4.16.3 Default Icons	C-3
28.4.16.4 Sample Icons	C-3
D. Free Menu	D-1
28.7 Free Menus	D-1
28.7.1 Making a Free Menu	D-1
28.7.2 Free Menu Formatting	D-1
28.7.3 Free Menu Descriptions	D-2
28.7.4 Free Menu Group Properties	D-7
28.7.5 Other Group Properties	D-8

28.7.6 Free Menu Items	D-8
28.7.7 Free Menu Item Description	D-8
28.7.8 Free Menu Item Properties	D-9
28.7.9 Mouse Properties	D-10
28.7.10 System Properties	D-10
28.7.11 Predefined Item Types	D-11
28.7.12 Free Menu Item Highlighting	D-14
28.7.13 Free Menu Item Links	D-14
28.7.14 Free Menu Window Properties	D-15
28.7.15 Free Menu Interface Functions	D-15
28.7.16 Accessing Functions	D-15
28.7.17 Changing Free Menus	D-16
28.7.18 Editor Functions	D-17
28.7.19 Miscellaneous Functions	D-18
28.7.20 Free Menu Macros	D-18
E. Error System	E-1
Summary of Error System Changes	E-1
Introduction to Error System Terminology	E-3
Program Interface to the Condition System	E-5
Defining and Creating Conditions	E-5
Signalling Conditions	E-8
Handling Conditions	E-11
Restarts	E-13
INDEX	INDEX-1

[This page intentionally left blank]

The *Lisp Release Notes* provide current information about the Lisp software development environment. You will find the following information in these *Notes*:

- An overview of significant extensions to the Common Lisp language.
- Descriptions of new features that enhance the integration and implementation of Common Lisp into the Lisp environment.
- A summary of changes made in the Library modules, in the Sketch and TEdit tools, and in the 1108 and 1186 User's Guides.
- Discussions of how specific Common Lisp features have affected the Interlisp-D language.
- Notes reflecting the changes made to Interlisp-D, independent of Common Lisp.

Contents are Organized

The *Lisp Release Notes* are organized as follows:

Chapter 1, Introduction, summarizes the Medley release enhancements.

Chapter 2, Notes and Cautions, highlights significant Medley and Lyric changes in the Lisp environment.

Chapter 3, Common Lisp/Interlisp-D Integration, discusses how the integration of Common Lisp into the Lisp environment affects Interlisp features.

Chapter 4, Changes to Interlisp-D in Lyric/Medley, outlines changes that have taken place in Interlisp-D and its environment during the Lyric and Medley releases. These changes are primarily independent of Common Lisp integration.

Chapters 3 and 4 are organized to parallel the *Interlisp-D Reference Manual* as closely as possible. To make it easy to use these chapters with the *IRM*, the following conventions are used:

Information is organized by *Interlisp-D Reference Manual* volume and section. The *IRM* section level headings are maintained to aid in cross-referencing.

Chapter 5, Library Modules, is a synopsis of the changes to the Lisp Library Modules.

Chapter 6, User's Guides, is a collection of release notes on the 1108 and 1186 User's Guides; *A User's Guide to Sketch*, and *A User's Guide to TEdit*.

Chapter 7, Common Lisp Implementation, describes improved features that integrate Common Lisp into the environment.

Five Appendices contain documentation of newly integrated system features:

Appendix A, The Exec, describes Lisp's Exec.

Appendix B, SEdit, describes the Lisp structure editor.

Appendix C, ICONW, describes the Lisp feature for building display icons.

Appendix D, Free Menu, describes Lisp's flexible menu feature.

Appendix E, Error System, describes error conditions and recovery.

Notational Conventions

Conventions used in the *Lisp Release Notes* include the following:

Names of Interlisp functions, macros and variables are shown in **BOLD UPPERCASE**; their arguments are in *ITALICS*.

Names of Common Lisp functions, macros and variables are shown in **bold lowercase**; their arguments are in *italics*.

A backslash (\) character preceding a function or variable name signifies that it is a property of the system.

Examples are shown in `terminal 10`.

Text shown with ~~StrikeThru~~ is information that no longer applies.

Text shown with revision bars in the right margin is information that has been added or modified since the last release.

References to the *Interlisp-D Reference Manual*, or *IRM*, are used throughout this manual.

How to Use The Release Notes

The *Lisp Release Notes* contain current information on the Lisp environment. The Medley release enhances the Lyric release with new features and corrections to over 450 known Lyric bugs. Because Medley primarily contains additions to Lyric, these *Release Notes* have been written to include Lyric information that applies in Medley.

These *Lisp Release Notes* replace the *Lyric Release Notes*. The descriptions contained within these *Notes* are closely interwoven with functions, variables and other concepts discussed in the *Interlisp-D Reference Manual*, or *IRM*. Chapters 3 and 4 of these *Notes* closely parallel the *IRM*, preserving section headings with respect to order and numbering. You might find it helpful to go through the *Release Notes* and the *IRM* together, marking the *IRM* sections that have new information. Later when you consult the *IRM* you will know which sections require you to read the analogous section of the *Release Notes*.

Related Literature

We recommend that you use the *Lisp Release Notes* as a supplement to the following publications:

Interlisp-D Reference Manual, Volumes I through III, Koto Release, 1985.

Common Lisp, the Language, by Guy L. Steele Jr., Digital Press, 1984.

Common Lisp Implementation Notes, Lyric Release, 1987.

Lisp Documentation Tools, (includes "A User's Guide to TEdit" and "A User's Guide to Sketch"), Lyric Release, 1987.

Lisp Library Modules, Medley Release, 1988.

Medley 1.0-S User's Guide, Medley Release, 1988.

[This page intentionally left blank]

The *Lisp Release Notes* contain information from both the Lyric and Medley releases, including descriptions of all Lyric bug fixes. Medley additions are indicated with revision bars in the right margin.

Summary of Medley Changes

The Medley release is currently provided on two platforms, Xerox 1100 series workstations using Medley 1.0, and Sun 3 workstations using Medley 1.0-S. Medley 1.0 and Medley 1.0-S are compatible with each other and will let you develop software on either platform. Source and compiled files are transferable between the two platforms. Sysouts developed on Xerox workstations can also be run on the Sun 3. Sysouts made on the Sun 3, however, cannot be run on Xerox workstations.

The Medley release enhances the Lyric release and fixes over 450 known Lyric bugs. Medley adds new features, improves Common Lisp implementation, and improves overall reliability of the Lisp sysout. Specific enhancements include:

- The COMPILER contains many added optimizations and numerous bug fixes.
- The DEBUGGER evaluates lexical variables. Lexical variables can now be contained in interpreted code.
- DFASL files now behave at the level of Interlisp-D compiled files. COMS are contained in DFASLs so that the system loads a DFASL file only once.
- The SEDIT code editor is more robust and better integrated with the environment.
- Common Lisp comments are preserved during loading. During MAKEFILE, comments can be written out with just semicolons.
- The new ERROR SYSTEM is compatible with the most recent standard defined for Common Lisp error systems.
- TEdit contains numerous bug fixes.
- MASTERSCOPE contains Common Lisp query support allowing you to ask questions about Common Lisp code that could previously be asked only of Interlisp-D code. Currently, questions specific to Common Lisp constructs are not supported.
- RS232 contains many bug fixes that improve the reliability of data transfer and the addition of various debugging tools.
- TCP/IP now contains many bug fixes including UNIX file interface and directory enumeration.

- A new System Tool lets you fetch sysouts from TCP hosts.
- NS Random Access is now supported.
- A new File Browser user interface now supports file sorting by dates. The new interface includes the ability to stop in the middle of operations.
- The Medley sysout is about the same size as the Lyric sysout.

In addition, Medley on the Sun 3 workstation offers the following new features:

- The UnixChat library module allows you to communicate with a UNIX shell on your own host.
- The UnixComm library module allows you to start a Unix process on a Sun workstation and provides an interface to the SunOS operating system.
- The ability to suspend Medley and use UNIX as a background process is provided.

[This page intentionally left blank]

This section contains notes and cautions that apply in Lyric and Medley. Medley notes are indicated with revision bars in the right margin. Text shown with ~~StrikeThru~~ is that information from the Lyric release that no longer applies.

Changes and Cautions in the Medley Release

- The Medley Release is currently provided on two platforms, the Xerox 1100 series workstations and selected Sun workstations. File structure for the 1108/09/86 remains the same. For Sun workstations, UNIX file structure is supported. See the *Medley 1.0-S User's Guide* for details.
- Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.
- SEdit and definers now support four-semicolon and balanced comments. Print support for these new types of comments is also provided. For details, see "TextModules" in the *Lisp Library Modules* manual, and "SEdit" in Appendix B of this manual.
- Medley and Lyric can both be installed on one machine.

Changes and Cautions in the Lyric Release

- Koto and Lyric cannot both be supported on one machine.
- You must have Services 10.0 installed on your printers to correctly print TEdit files.
- Interlisp DMACROs are not visible to Common Lisp. If a symbol has both a function definition and a DMACRO property, the compiler assumes that the DMACRO is an optimizer for the old Interlisp compiler and ignores it.
- The Common Lisp functions found in *Common Lisp, The Language*, section 25.4.2, "Other environmental inquiries" (e.g., LISP-IMPLEMENTATION-TYPE) are in the COMMON LISP (CL:) package.
- Both Medley and Lyric use the new type of Executive, and both sysouts contain the ability to spawn multiple executive processes. The default executive is Common Lisp, not Interlisp. The old Executive (the "Programmer's Assistant") is not available in Medley.

You should be particularly careful in the new Executives when typing file names, as some file name delimiters now have syntactic significance in the new readtables. In particular, the character colon (:) used in NS file server names is a package delimiter in all new Executives, and the version delimiter semi-colon (;) is a comment character in the Common Lisp Executives. If you type a file name in the form of a symbol to an Exec, you must escape the special characters, or use the multiple escape character around the whole name. For example, in a Common Lisp Exec you might type

`{FS:Me:Company}<Fred>Stuff.tedit;3`

or

`|{FS:Me:Company}<Fred>Stuff.tedit;3|,`

which are equivalent, except that the former is read as all upper case (Common Lisp Exec's read case-insensitively). This caution should also be noted when copy-selecting file names out of a File Browser.

It is recommended that you type file names as strings whenever possible, as virtually all system interfaces accept strings instead of symbols. Two notable exceptions are MAKEFILE and TEDIT, which require symbols when naming files.

These escaping rules apply *only* to file names typed to an Executive (or in general, a Lisp reader). Individual tools that prompt for a file name read the name as a string, so escape characters need not (and should not) be typed. In particular, this is true for the prompt windows of TEdit and File Browser, and the prompt for an Init file when a system with no local Init file is started up.

- A new error system, based on the current Common Lisp proposed error standard, replaces the old Interlisp error system.
- The **!EVAL** debugger command no longer exists and the = and -> break commands are no longer supported..
- The function **ERRORN** no longer exists and **ERRORTYPELIST** is no longer supported. See Chapter 3, Common Lisp /Interlisp Integration, section 14.10 "Error List" for Interlisp errors that are no longer supported.
- The Lyric release contained a new compiler and compiled code format, .DFASL (FASt Loading) files. The old compiler is still available and produces files in the old format, but with extension .LCOM. The old compiler will not be available in future releases.
- Files produced by the Lyric File Manager cannot be loaded into previous releases of the system. Files compiled in Koto cannot be loaded into Lyric.
- SETQ from the exec does not interact with the File Manager, nor does it print (var reset) (except in the "Programmer's Assistant").

- DWIM/CLISP: CLISP infix is no longer fully supported; users should dwimify old Koto code before running it in Lyric. Additionally, WITH constructs using " \leftarrow " and BIND constructs in the form of an atom $A \leftarrow B$ need to be dwimified.
- The functions BREAKDOWN and BRKDOWNRESULTS as well as the variables, BRKDWNTYPE and BRKDWNTYPES have been removed from the environment. The Lisp Library Module, SPY supersedes BREAKDOWN.
- The file system supports having multiple streams opened on a single file at one time. This means that the input/output functions accept only streams as arguments, not symbols naming files. This has several implications for Interlisp programmers, one being that the function **CLOSEALL** is no longer implemented. See the Chapter 3, Common Lisp/Interlisp Integration, Streams and Files section, for details.
- Windows cannot be used interchangeably with streams in Common Lisp functions. If you need to use a window in the middle of a Common Lisp function, use (IL:GETSTREAM window) to get the associated display stream.
- Loading CPM-format floppies is very slow in Lyric. CPM-format floppies are not supported in Medley.
- The default Interlisp readtable has been modified for compatibility with Common Lisp. The characters colon (:), hash (#) and vertical bar (|) have different meaning. The File Manager gives a choice of reader environments in which to write files, and remembers which one was used for each file.
- READ/PRINT consistency: Old Interlisp code that used **READ** and **PRINT** without being careful about using a particular readtable may need to be fixed.
- The Interlisp function **SKREAD** defaults its readtable argument to the current readtable, viz., the value of ***READTABLE***, rather than **FILERDTBL**.
- FREEMENU and ICONW, formerly Library modules, are included in the Lisp.sysout in Lyric and Medley.
- The Lyric Lisp editor, SEdit, has been modified in Medley. DEdit is now a library module.
- Revised fonts: Lyric revised the naming convention for font files, and printer width files had corrected line leading information. ~~Old Koto fonts can still be used, but you are encouraged to start using the new fonts as soon as practicable.~~ Medley and Lyric fonts are completely compatible.
- Lyric image objects are now stored on files in a way that cannot always be read into Koto. (Lyric, on the other hand, can read Koto image objects.) This means, for example, that you may not be able to share TEdit files or sketches containing image objects between Koto and Lyric.
- The field names for the CURSOR datatype have been changed.

- Masterscope has been removed from the standard environment. If you wish to use it, load the Masterscope Library module.
- Pattern matching is no longer a part of the standard environment. Pattern matching can be found in the Lisp Library Module, Match.
- PRESS fonts are not part of the standard Lisp environment. PRESS is now available as a Library Module.
- In Lyric, the Library module TCP/IP does not work on 1186 workstations that have *both* IOPs with part number 140K03030 *and* "old" ROMs. The problem is not with the IOP board per se, rather it's a problem with the IOP's ROMs. If TCP/IP doesn't work on your 1186 you should check your IOP board revision. If you have the old IOP you may need to replace the ROMs before you can use TCP/IP, contact your service representative.

TCP/IP does work with newer IOPs—part number 140K05560.

If you attempt to Teleraid a Lyric sysout from a Koto sysout, you should be aware of the following:

1. All symbols will be read as if they were in the INTERLISP package and you can only type a subset of the IL symbols to it.
2. Teleraid will not understand certain Common Lisp datatypes, such as CHARACTER and strings.

[This page intentionally blank]

3. COMMON LISP/INTERLISP-D INTEGRATION

NOTE: Chapter 3 is organized to correspond to the original *Interlisp-D Reference Manual*, and explains changes related to how Common Lisp affects Interlisp-D in your Lisp software development environment. To make it easy to use this chapter with the *IRM*, information is organized by *IRM* volume and section numbers. Section headings from the *IRM* are maintained to aid in cross-referencing.

Lyric information as well as Medley release enhancements are included. Medley additions are indicated with revision bars in the right margin.

VOLUME I—LANGUAGE

Chapter 2 Litatoms

(2.1)

What Interlisp calls a "LITATOM" is the same as what Common Lisp calls a "SYMBOL." Symbols are partitioned into separate name spaces called packages. When you type a string of characters, the resulting symbol is searched for in the "current package." A colon in the symbol separates a package name from a symbol name; for example, the string of characters "CL:AREF" denotes the symbol AREF accessible in the package CL. For a full discussion, see Guy Steele's *Common Lisp, the Language*.

All the functions in this section that create symbols do so in the INTERLISP package (IL), which is also where all the symbols in the *Interlisp-D Reference Manual* are found. Note that this is true even in cases where you might not expect it. For example, U-CASE returns a symbol in the INTERLISP package, even when its argument is in some other package; similarly with L-CASE and SUBATOM. In most cases, this is the right thing for an Interlisp program; e.g., U-CASE in some sense returns a "canonical" symbol that one might pass to a SELECTQ, regardless of which executive it was typed in. However, to perform symbol manipulations that preserve package information, you should use the appropriate Common Lisp functions (See *Common Lisp the Language*, Chapter 11, Packages and Chapter 18, Strings).

Symbols read under an old Interlisp readtable are also searched for in the INTERLISP package. See Section 25.8, Readtables, for more details.

Section 2.1 Using Litatoms as Variables

(I:2.3)

(**BOUNDP** *VAR*)

[Function]

The Interlisp interpreter has been modified to consider any symbol bound to the distinguished symbol **NOBIND** to be unbound. It will signal an UNBOUND-VARIABLE condition on encountering references to such symbols. In prior releases, the interpreter only considered a symbol unbound if it had no dynamic binding and in addition its top-level value was **NOBIND**.

For most user code, this change has no effect, as it is unusual to bind a variable to the particular value **NOBIND** and still deliberately want the variable to be considered bound. However, it is a particular problem when an interpreted Interlisp function is passed to the function **MAPATOMS**. Since **NOBIND** is a symbol, it will eventually be passed as an argument to the interpreted function. The first reference to that argument within the function will signal an error.

A work-around for this problem is to use a Common Lisp function instead. Calls to this function will invoke the Common Lisp interpreter which will treat the argument as a local, not special, variable. Thus, no error will be signaled. Alternatively, one could include the argument to the Interlisp function in a **LOCALVARS** declaration and then compile the function before passing it to **MAPATOMS**. This has the advantage of significantly speeding up the **MAPATOMS** call.

Section 2.3 Property Lists

(I:2.6)

The value returned from the function **REMPROP** has been changed in one case:

(**REMPROP** *ATM PROP*)

[Function]

Removes all occurrences of the property *PROP* (and its value) from the property list of *ATM*. Returns *PROP* if any were found (**T** if *PROP* is **NIL**), otherwise **NIL**.

Section 2.4 Print Names

(I:2.7)

The print functions now qualify the name of a symbol with a package prefix if the symbol is not accessible in the current package. The Interlisp "PRIN1" print name of a symbol does not include the package name.

(I:2.10)

The **GENSYM** function in Interlisp creates symbols interned in the INTERLISP package. The Common Lisp **CL:GENSYM** function creates uninterned symbols.

*(l:2.11)***(MAPATOMS FN)**

[Function]

See the note for **BOUNDP** above.

Section 2.5 Characters

A "character" in Interlisp is different from the type "character" in Common Lisp. In Common Lisp, "character" is a distinguished data type satisfying the predicate **CL:CHARACTERP**. In Interlisp, a "character" is a single-character symbol, not distinguishable from the type symbol (litatom). Interlisp also uses a more efficient object termed "character code", which is indistinguishable from the type integer.

Interlisp functions that take as an argument a "character" or "character code" do not in general accept Common Lisp characters. Similarly, an Interlisp "character" or "character code" is not acceptable to a Common Lisp function that operates on characters. However, since Common Lisp characters are a distinguished datatype, Interlisp string-manipulation functions are willing to accept them any place that a "string or symbol" is acceptable; the character object is treated as a single-character string.

To convert an Interlisp character code *n* to a Common Lisp character, evaluate (**CL:CODE-CHAR** *n*). To convert a Common Lisp character to an Interlisp character code, evaluate (**CL:CHAR-CODE** *n*). For character literals, where in Interlisp one would write (**CHARCODE** *x*), to get the equivalent Common Lisp character one writes #\x. In this syntax, *x* can be any character or string acceptable to **CHARCODE**; e.g., #\GREEK-A.

Chapter 4 Strings

(l:4.1)

Interlisp strings are a subtype of Common Lisp strings. The functions in this chapter accept Common Lisp strings, and produce strings that can be passed to Common Lisp string manipulation functions.

Chapter 5 Arrays

Interlisp arrays and Common Lisp arrays are disjoint data types. Interlisp arrays are not acceptable arguments to Common Lisp array functions, and vice versa. There are no functions that convert between the two kinds of arrays.

Chapter 6 Hash Arrays

Interlisp hash arrays and Common Lisp hash tables are the same data type, so Interlisp and Common Lisp hash array functions may be freely intermixed. However, some of the arguments are different; e.g., the order of arguments to the map functions in **IL:MAPHASH** and **CL:MAPHASH** differ. The extra functionality of specifying your own hashing function is available only from Interlisp **HASHARRAY**, not **CL:MAKE-HASH-TABLE**, though the latter does supply the three built-in types specified by *Common Lisp, the Language*.

Chapter 7 Numbers and Arithmetic Functions

(I:7.2)

The addition of Common Lisp data structures within the Lisp environment means that there are some invariants which used to be true for anything in the environment that are no longer true.

For example, in Interlisp, there were two kinds of numbers: integer and floating. With Common Lisp, there are additional kinds of numbers, namely ratios and complex numbers, both of which satisfy the Interlisp predicate **NUMBERP**. Thus, **NUMBERP** is no longer the simple union of **FIXP** and **FLOATP**. It used to be that a program containing

```
(if (NUMBERP X)
    then (if (FIXP X)
              then ...assume X is an integer ...
              else ...can assume X is floating point...))
```

would be correct in Interlisp. However, this is no longer true; this program will not deal correctly with ratios or complex numbers, which are **NUMBERP** but neither **FIXP** nor **FLOATP**.

Section 7.2 Integer Arithmetic

When typing to a *new* Interlisp Executive, the input syntax for integers of radix other than 8 or 10 has been changed to match that of Common Lisp. Use # instead of |, e.g., #b10101 is the new syntax for binary numbers, #x1A90 for hexadecimal, etc. Suffix Q is still recognized as specifying octal radix, but you can also use Common Lisp's #o syntax.

(I:7.4)

In the Lyric release, the FASL machinery would handle some positive literals incorrectly, reading them back as negative numbers. The numbers handled incorrectly were those numbers x greater than 2^{31-1} for which $(\text{mod}(\text{integer-length } x) 8)$ was zero. The Medley release fixes this situation. Any files containing such numbers should be recompiled.

Chapter 10 Function Definition, Manipulation, and Evaluation

Section 10.1 Function Types

All Interlisp **NLAMBDAs** appear to be macros from Common Lisp's point of view. This is discussed at greater length in *Common Lisp Implementation Notes*, Chapter 8, Macros.

Section 10.6 Macros

(**EXPANDMACRO** *EXP QUIETFLG* — —)

[Function]

EXPANDMACRO only works on Interlisp macros, those appearing on the **MACRO**, **BYTEMACRO** or **DMACRO** properties of symbols. Use **CL:MACROEXPAND-1** to expand Common Lisp macros and those Interlisp macros that are visible to the Common Lisp compiler and interpreter.

Section 10.6.1 DEFMACRO

(I:10.24)

Common Lisp does not permit a symbol to simultaneously name a function and a macro. In Lyric, this restriction also applies to Interlisp macros defined by **DEFMACRO**. That is, evaluating **DEFMACRO** for a symbol automatically removes any function definition for the symbol. Thus, if your purpose for using a macro is to make a function compile in a special way, you should instead use the new form **XCL:DEFOPTIMIZER**, which affects only compilation. The *Xerox Common Lisp Implementation Notes* describe **XCL:DEFOPTIMIZER**.

Interlisp **DMACRO** properties have typically been used for implementation-specific optimizations. They are not subject to the above restriction on function definition. However, if a symbol has both a function definition and a **DMACRO** property, the Lisp compiler assumes that the **DMACRO** was intended as an optimizer for the old Interlisp compiler and ignores it.

Chapter 11 Stack Functions

Section 11.1 The Spaghetti Stack

Stack pointers now print in the form

#<Stackp address/frame-name>.

Some restrictions were placed on spaghetti stack manipulations in order to integrate reasonably with Common Lisp's **CL:CATCH** and **CL:THROW**. In Lyric, it is an error to return to the same frame twice, or to return to a frame that has been unwound through. This means, for example, that if you save a stack pointer to one of your ancestor frames, then perform a **CL:THROW** or **RETFROM** that returns "around" that frame, i.e., to an ancestor of that frame, then

the stack pointer is no longer valid, and any attempt to use it signals an error "Stack Pointer has been released". It is also an error to attempt to return to a frame in a different process, using **RETFROM**, **RETTO**, etc.

The existence of spaghetti stacks raises the issue of under what circumstances the cleanup forms of **CL:UNWIND-PROTECT** are performed. In Lisp, **CL:THROW** always runs the cleanup forms of any **CL:UNWIND-PROTECT** it passes. Thanks to the integration of **CL:UNWIND-PROTECT** with **RESETLST** and the other Interlisp context-saving functions, **CL:THROW** also runs the cleanup forms of any **RESETLST** it passes. The Interlisp control transfer constructs **RETFROM**, **RETTO**, **RETEVAL** and **RETAPPLY** also run the cleanup forms in the analogous case, viz., when returning to a direct ancestor of the current frame. This is a significant improvement over prior releases, where **RETFROM** never ran any cleanup forms at all.

In the case of **RETFROM**, etc, returning to a non-ancestor, the cleanup forms are run for any frames that are being abandoned as a result of transferring control to the other stack control chain. However, this should not be relied on, as the frames would not be abandoned at that time if someone else happened to retain a pointer to the caller's control chain, but subsequently never returned to the frame held by the pointer. Cleanup forms are *not* run for frames abandoned when a stack pointer is released, either explicitly or by being garbage-collected. Cleanup forms are also not run for frames abandoned because of a control transfer via **ENVEVAL** or **ENVAPPLY**. Callers of **ENVEVAL** or **ENVAPPLY** should consider whether their intent would be served as well by **RETEVAL** or **RETAPPLY**, which *do* run cleanup forms in most cases.

Chapter 12 Miscellaneous

Section 12.4 System Version Information

All the functions listed on page 12.12 in the *Interlisp-D Reference Manual* have had their symbols moved to the LISP (CL) package. They are *not* shared with the INTERLISP package and any references to them in your code will need to be qualified i.e., **CL:name**.

Section 12.8 Pattern Matching

Pattern matching is no longer a standard part of the environment. The functionality for Pattern matching can be found in the Lisp Library Module called **MATCH**.

[This page intentionally left blank]

VOLUME II—ENVIRONMENT

Chapter 13 Interlisp Executive

[This chapter of the *Interlisp-D Reference Manual* has been renamed Chapter 13, Executives.]

Lisp has a new kind of Executive (or Exec), designed for use in an environment with both Interlisp and Common Lisp. This executive is available in three standard modes, distinguished by their default settings for package and readtable:

XCL	New Exec. Uses XCL readtable, XCL-USER package
CL	New Exec. Uses LISP readtable, USER package
IL	New Exec. Uses INTERLISP readtable, INTERLISP package

In addition, the old Interlisp executive, the "Programmer's Assistant", is still available in this release for the convenience of Koto users:

OLD-INTERLISP	Old "Programmer's Assistant" Exec. Uses OLD-INTERLISP-T readtable, INTERLISP package. It is likely that this executive will not be supported in future releases.
---------------	--

When Lisp starts, it is running a single executive, the XCL Exec. You can spawn additional executives by selecting EXEC from the background menu. The type of an executive is indicated in the title of its window; e.g., the initial executive has title "Exec (XCL)". Each executive runs in its own process; when you are finished with an executive, you can simply close its window, and the process is killed.

The new executive is modeled, somewhat, on the old "Programmer's Assistant" executive and, to a first approximation, you can type to it just as you did in past releases. You should note, however, that the default executive (XCL) expects Common Lisp input syntax, and reads symbols relative to the XCL-USER package. This means that to type Interlisp symbols, you must prefix the symbol with the characters "IL:" (in upper or lower case). And even in the new IL executive, the readtable being used is the new INTERLISP readtable, in which the characters colon (:), vertical bar (|) and hash (#) all have different meanings than in Koto.

The OLD-INTERLISP exec, with one exception, uses exactly the same input syntax as in Koto; this means in particular that colon cannot be used to type package-qualified symbols, since colon is an ordinary character there. The one exception is that there *is* a package delimiter character in the OLD-INTERLISP readtable, should you have a need to use it—Control-↑, which usually echoes as "↑↑", though it may appear as a black rectangle in some fonts.

The new executive does differ from the old one in several respects, especially in terms of its programmatic interface. Complete details

of the new executive can be found in Appendix A. The Exec. Some of the important differences are:

- Executives are numbered

Executives, other than the first one, are labeled with a distinct number. This number appears in the exec window's title, and also in its prompt, next to the event number. The OLD-INTERLISP executive does not include this exec number.

- Event number allocation

The numbers for events are allocated at the time the prompt for the event is printed, but all execs still share a common event number space and history list. This means that ?? shows all events that have occurred in *any* executive, though not necessarily in the order in which the events actually occurred (since it is the order in which the event numbers were allocated). Events for which the type-in has not been completed are labeled "<in progress>" in the ?? listing. In the old executive, event numbers are not allocated until type-in is complete, which means that the number printed next to the prompt is not necessarily the number associated with the event, in the case that there has been activity in other executives.

In the new executive, relative event specifications are local to the exec; e.g., -1 refers to the most recent event in that specific exec. In the old executive, -1 referred to the immediately preceding event in *any* executive.

- New facility for commands

The old Executive has commands based on **LISPMACROS**. The new Executive has its own command facility, **XCL:DEFCOMMAND**, which allows commands to be named without regard to package, and to be written with familiar Common Lisp style of argument list.

- Commands are typed *without* parentheses

In the old executive, a command could be typed with or without enclosing parentheses. In the new executive, a parenthesized form is always interpreted as an EVAL-style input, never a command.

- **SETQ** does not interact with the File Manager

In the Koto release, when you typed in the Exec

(SETQ FOO some-new-value-for-FOO)

the executive responded (**FOO reset**), and the file package was told that **FOO**'s value changed. Any files on which **FOO** appeared as a variable would then be marked as needing to be cleaned up. If **FOO** appeared on no file, you'd be prompted to put it on one when you ran (**FILES?**).

This is still the case in the old executive. However, it is no longer the case in the new executive. If you are setting a variable that is significant to a program and you want to save it on a file, you should use the Common Lisp macro **CL:DEFPARAMETER** instead of **SETQ**. This will give the symbol a definition of type **VARIABLES** (rather than **VARS**), and it will be noticed by the File manager. If you want to change the value of the variable, you must either use

CL:DEFPARAMETER again, or edit the variable using **ED** (not **DV**).

- Programmatic interface completely different

As a first approximation, all the functions and variables in *IRM* Sections 13.3 (except the **LISPXPRINT** family) and 13.6 apply only to the Old Interlisp Executive, unless specified otherwise in Appendix A. In particular, the variables **PROMPT#FLG**, **PROMPTCHARFORMS**, **SYSPRETTYFLG**, **HISTORYSAVEFORMS**, **RESETFORMS**, **ARCHIVEFN**, **ARCHIVEFLG**, **LISPXUSERFN**, **LISPMACROS**, **LISPMHISTORYMACROS** and **READBUF** are not used by the new Exec. The function **USEREXEC** invokes an old-style Executive, but uses the package and readtable of its caller. The function **LISPXUNREAD** has no effect on the new Exec. Callers of **LISPEVAL** are encouraged to use **EXEC-EVAL** instead.

Some subsystems still use the old-style Executive—in particular, the tty structure editor.

Chapter 14 Errors and Breaks

Lisp extends the Interlisp break package to support multiple values and the Common Lisp lambda syntax. Interlisp errors have been converted to Common Lisp conditions.

Note that Sections 14.2 through 14.6 in the *Interlisp-D Reference Manual* have been replaced by new Debugger information (see *Common Lisp Implementation Notes*).

Section 14.3 Break Commands

(II:14.6)

The **!EVAL** debugger command no longer exists.

(II:14.10-11)

The Break Commands = and -> are no longer supported.

Section 14.6 Creating Breaks with **BREAK1**

While the function **BREAK1** still exists, broken and traced functions are no longer redefined in terms of it. More primitive constructs are used.

Section 14.7 Signalling Errors

Interlisp errors now use the new XCL error system. Most of the functions still exist for compatibility with existing Interlisp code, but the underlying machinery is different. There are some incompatible differences, however, especially with respect to error numbers.

The old Interlisp error system had a set of registered error numbers for well known error conditions, and all other errors were identified

by a string (an error message). In the new Lisp error system, all errors are handled by signalling an object of type **XCL:CONDITION**. The mapping from Interlisp error numbers to Lisp conditions is given below in Section 14.10.

Since one cannot in general map a condition object to an Interlisp error number, the function **ERRORN** no longer exists. The equivalent functionality exists by examining the special variable ***LAST-CONDITION***, whose value is the condition object most recently signaled.

(ERRORX ERXM) calls **CL:ERROR** after first converting **ERXM** into a condition in the following way: If **ERXM** is **NIL**, the value of ***LAST-CONDITION*** is used; if **ERXM** is an Interlisp error descriptor, it is first converted to a condition; finally, if **ERXM** is already a condition, it is passed along unchanged. **ERRORX** also sets up a proceed case for **XCL:PROCEED**, which will attempt to re-evaluate the caller of **ERRORX**, much as **OK** did in the old Interlisp break package.

ERROR, **HELP**, **SHOULDNT**, **RESET**, **ERRORMESS**, **ERRORMESS1**, and **ERRORSTRING** work as before. All output is directed to ***ERROR-OUTPUT***, initially the terminal.

ERROR! is equivalent to the new error system's **XCL:ABORT** proceed function, except that if no **ERRORSET** or **XCL:CATCH-ABORT** is found, it unwinds all the way to the top of the process.

SETERRORN converts its arguments into a condition, then sets the value of ***LAST-CONDITION*** to the result.

Section 14.8 Catching Errors

ERRORSET, **ERSETQ** and **NLSETQ** have been reimplemented in terms of the new error system, but their behavior is essentially the same as before. **NLSETQ** catches all errors (conditions of type **CL:ERROR** and its descendants), and sets up a proceed case for **XCL:ABORT** so that **ERROR!** will return from it. **ERSETQ** also sets up a proceed case for **XCL:ABORT**, though it does not catch errors.

One consequence of the new implementation is that there are no longer frames named **ERRORSET** on the stack; programs that explicitly searched for such frames will have to be changed.

ERRORTYPELIST is no longer supported. The equivalent functionality is provided by default handlers. Although condition handlers provide a more powerful mechanism for programmatically responding to an error condition, old **ERRORTYPELIST** entries generally cannot be translated directly. Condition handlers that want to resume a computation (rather than, say, abort from a well-know stack location) generally require the cooperation of a proceed case in the signalling code; there is no easy way to provide a substitute value for the "culprit" to be re-evaluated in a general way.

One important difference between **ERRORTYPELIST** and condition handlers is their behavior with respect to **NLSETQ**. In Koto, the relevant error handler on **ERRORTYPELIST** would be tried, even for errors occurring underneath an **NLSETQ**. In Lyric, the **NLSETQ** traps all errors before the default condition handlers can see the

error. This means, for example, that the behavior of **(NLSETQ (OPENSTREAM --))** is now different if the **OPENSTREAM** causes a file not found error—in Koto, the system would search **DIRECTORIES** for the file; in Lyric, the **NLSETQ** returns **NIL** immediately without searching, since the default handler for **XCL:FILE-NOT-FOUND** is not invoked.

Section 14.9 Changing and Restoring System State

The special forms **RESETLST**, **RESETSAVE**, **RESETVAR**, **RESETVARS** and **RESETFORM** still exist, but are implemented by a new mechanism that also supports Common Lisp's **CL:UNWIND-PROTECT**. Common Lisp's **CL:THROW** and (in most cases) Interlisp's **RETFROM** and related control transfer constructs cause the cleanup forms of both **CL:UNWIND-PROTECT** and **RESETLST** (etc) to be performed. This is discussed in more detail in the notes for Chapter 11, the stack.

Section 14.10 Error List

Most of the Interlisp errors are mapped into condition types in Lisp. Some are no longer supported. Following is the list of error type mappings. The first name is the condition type that the error descriptor turns into. If there is a second name, it is the slot whose value is set to **CADR** of the error descriptor. Any additional pairs of items are the values of other slots set by the mapping. Attempting to use an unsupported error type number will result in a simple error to that effect.

- 0 Obsolete
- 1 Obsolete
- 2 **STACK-OVERFLOW**
- 3 **ILLEGAL-RETURN**
- 4 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT:EXPECTED-TYPE 'LIST*
- 5 **XCL:SIMPLE-DEVICE-ERROR** *MESSAGE*
- 6 **XCL:ATTEMPT-TO-CHANGE-CONSTANT**
- 7 **XCL:ATTEMPT-TO-RPLAC-NIL** *MESSAGE*
- 8 **ILLEGAL-GO TAG**
- 9 **XCL:FILE-WONT-OPEN** *PATHNAME*
- 10 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT:EXPECTED-TYPE 'CL:NUMBER*
- 11 **XCL:SYMBOL-NAME-TOO-LONG**
- 12 **XCL:SYMBOL-HT-FULL**
- 13 **XCL:STREAM-NOT-OPEN** *STREAM*
- 14 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT:EXPECTED-TYPE 'CL:SYMBOL*
- 15 Obsolete
- 16 **END-OF-FILE** *STREAM*
- 17 **INTERLISP-ERROR** *MESSAGE*
- 18 Not supported (control-B interrupt)

- 19 **ILLEGAL-STACK-ARG** *ARG*
- 20 Obsolete
- 21 **XCL:ARRAY-SPACE-FULL**
- 22 **XCL:FS-RESOURCES-EXCEEDED**
- 23 **XCL:FILE-NOT-FOUND** *PATHNAME*
- 24 Obsolete
- 25 **INVALID-ARGUMENT-LIST** *ARGUMENT*
- 26 **XCL:HASH-TABLE-FULL** *TABLE*
- 27 **INVALID-ARGUMENT-LIST** *ARGUMENT*
- 28 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE**
'ARRAYP
- 29 Obsolete
- 30 **STACK-POINTER-RELEASED** *NAME*
- 31 **XCL:STORAGE-EXHAUSTED**
- 32 Not supported (attempt to use item of incorrect type)
- 33 Not supported (illegal data type number)
- 34 **XCL:DATA-TYPES-EXHAUSTED**
- 35 **XCL:ATTEMPT-TO-CHANGE-CONSTANT**
- 36 Obsolete
- 37 Obsolete
- 38 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE**
'READTABLEP
- 39 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE**
'TERMTABLEP
- 40 Obsolete
- 41 **XCL:FS-PROTECTION-VIOLATION**
- 42 **XCL:INVALID-PATHNAME** *PATHNAME*
- 43 Not supported (user break)
- 44 **UNBOUND-VARIABLE** *NAME*
- 45 **UNDEFINED-CAR-OF-FORM** *FUNCTION*
- 46 **UNDEFINED-FUNCTION-IN-APPLY**
- 47 **XCL:CONTROL-E-INTERRUPT**
- 48 **XCL:FLOATING-UNDERFLOW**
- 49 **XCL:FLOATING-OVERFLOW**
- 50 Not supported (integer overflow)
- 51 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE**
'CL:HASH-TABLE
- 52 **TOO-MANY-ARGUMENTS** *CALLEE* **:MAXIMUM CL:CALL-**
ARGUMENTS-LIMIT

Note that there are many other condition types in Lisp; see the error system documentation in the *Common Lisp Implementation Notes* for details.

Chapter 15 Breaking Functions and Debugging

In Lyric the uses of **BREAK**, **TRACE**, and **ADVISE** are unchanged, from the user's point of view, but the internals of their implementation are quite different.

For complete documentation on the new implementation of breaking, tracing and advising, see the *Common Lisp Implementation Notes*, Section 25.3.

In particular, you should note the following differences:

- The variable **BRKINFOLST** no longer exists and the format of the value of the variable **BROKENFNFS** has changed. In addition, the **BRKINFO** property is no longer used.
- **BREAK** and **TRACE** no longer work on CLISP words.
- The **BREAKIN** and **UNBREAKIN** functions no longer exist. No comparable facility exists in Lisp. The user can manually insert calls to the Common Lisp function **CL:BREAK** in order to create a breakpoint at that point in the function.

Please note the following additional changes to breaking functions:

Section 15.1 Breaking Functions and Debugging

(BREAK0 FN WHEN COMS — —)

[Function]

The function **BREAK0** now works when applied to an undefined function. This allows you to use the breaking facility to create "stubs" that generate a breakpoint when called. You can then examine the arguments passed and use the **RETURN** command in the debugger to return the proper result(s).

The "break commands" facility (the **COMS** argument) is no longer supported. **BREAK0** now signals an error when supplied with a non-**NIL** third argument. If you need finer control over the functioning of breakpoints you are directed to the **ADVISE** facility; it offers complete control of how and when the given function is evaluated.

Passing a non-atomic argument in the form **(FN1 IN FN2)** as the first argument to **BREAK0** still has the effect of creating a breakpoint wherever **FN2** calls **FN1**. However, it no longer creates a function named **FN1-IN-FN2** to do so. In addition, the format of the value of the **NAMESCHANGED** property has changed and the **ALIAS** property is no longer used.

(TRACE X)

[Function]

TRACE is no longer a special case of **BREAK**, though the functions **UNBREAK** and **REBREAK** continue to work on traced functions.

In addition, the function **TRACE** no longer calls **BREAK0** in order to do its job. Also, non-atomic arguments to **TRACE** no longer specify forms the user wishes to see in the tracing output.

(UNBREAK X)

[Function]

The function **UNBREAK** is no longer implemented in terms of **UNBREAK0**, although that function continues to exist.

Section 15.2 Advising

The implementation of advising has been completely reworked. While the semantics implied by the code shown in Section 15.2.1 of the *Interlisp-D Reference Manual* is still supported, the details are quite different. In particular, it is now possible to advise functions that return multiple values and for **AFTER**-style advice to access those values. Also, all advice is now compiled, rather than interpreted. The advising facility no longer makes use of the special forms **ADV-PROG**, **ADV-RETURN**, and **ADV-SETQ**.

You should also note the following changes to the advise facility:

- The editing of advice has changed slightly. In previous releases, the advice and original function-body were edited simultaneously. In Lyric, they can only be edited separately. When you finish editing the advice for a function, that function is automatically re-advised using the new advice.
- The variable **ADVINFOLST** no longer exists and the format of the value of the variable **ADVISEDFNS** has changed. In addition, the properties **ADVICE** and **READVICE** are no longer used, except in the handling of advice saved on files from previous releases. Advice saved in Lyric does not use the **READVICE** property.
- The function **ADVISEDUMP** no longer exists.
- Advice saved on files in previous releases can, in general, be loaded into the Lyric system compatibly. A known exception is the case in which a list of the form **(FN1 IN FN2)** was given to the **ADVICE** or **ADVISE** file package commands. When **READVICE** is called on such a name, the old-style advice, on the **READVICE** property of the symbol **FN1-IN-FN2**, will not be found. This will eventually lead to an **XCL:ATTEMPT-TO-RPLAC-NIL** error. The user should evaluate the form
(RETFROM 'READVICE1)
in the debugger to proceed from the error and later evaluate
(READVICE FN1-IN-FN2)
by hand to install the advice.

- The **ADVISE** and **ADVISE** File Manager commands now accept three kinds of arguments:
 - a symbol, naming an advised function,
 - a list in the form **(FN1 :IN FN2)**, and
 - a symbol of the form **FN1-IN-FN2**.Arguments of the form **(FN1 IN FN2)** are not acceptable any longer. Arguments of the form **FN1-IN-FN2** should be converted into the equivalent form **(FN1 :IN FN2)**.

(ADVISE WHO WHEN WHERE WHAT)

[Function]

In the Lyric release of Lisp, **ADVISE** has some changes in the way arguments are treated and the possible values for those arguments. Most notably:

- In earlier releases, you could call **ADVISE** with only one argument, the name of a function. In this case, **ADVISE** "set up" the named function for advising, but installed no advice. This usage is no longer supported.
- Previously, an undocumented value of **BIND** was accepted for the **WHEN** argument to **ADVISE**. This kind of advice is no longer supported. It can be adequately simulated using **AROUND** advice.

In addition, advising Common Lisp functions works somewhat differently with respect to a function's arguments. The arguments are not available by name. Instead, the variable **XCL:ARGLIST** is bound to a list of the values passed to the function and may be changed to affect what will be passed on.

As with the breaking facility (see above), **ADVISE** no longer creates a function named **FN1-IN-FN2** as a part of advising **(FN1 IN FN2)**.

Chapter 16 List Structure Editor

The list structure editor, DEdit, is not part of the Lisp environment. It is now a Lisp Library Module. Chapter 16 has been renamed Structure Editor.

SEdit, the new Lisp editor, replaced DEdit in the Lyric release. The description of SEdit may be found in Appendix B of this volume. The commands used to invoke both SEdit and DEdit are the same.

Following is a description of the interface to the Lisp editor.

Switching Between Editors

If you have both SEdit and DEdit loaded, you can switch between them by calling: **(EDITMODE 'EDITORNAME)** where *EDITORNAME* is one of the symbols SEdit or DEdit.

Packages

The **ED** editor interface accepts TYPE information from the Interlisp or Common Lisp packages.

Starting a Lisp Editor

In the XCL environment, calling ED with a pathname will start the editor on the coms of the file (as if DC had been called).

<u>(ED NAME &OPTIONAL OPTIONS)</u>	<u>[Function]</u>
---	-------------------

This function starts the Lisp editor. **ED** is the default interface to the editor. SEdit is the default Lisp editor. The same symbol, **ED**, is exported in both the IL and CL packages.

NAME is the name of any File Manager object.

OPTIONS is either a single symbol or a list of symbols, each of which is either a File Manager type or one or more of the keywords **:DISPLAY**, **:DONTWAIT**, **:CURRENT**, **:COMPILE-ON-COMPLETION**, **:CLOSE-ON-COMPLETION**, or **:NEW**. If exactly one File Manager type is given, **ED** tries to edit that type of definition for *NAME*. If more than one type is given in *OPTIONS*, **ED** will determine for which of them *NAME* has a definition. If a definition exists for more than one of the types, **ED** gives you a choice of which one to edit. If no File Manager types are given, **ED** treats *OPTIONS* as a list of all of the existing types; thus you are given a choice of all of the existing definitions of *NAME*.

The variable **FILEPKGTYPES** contains a complete list of the currently-known manager types.

If the keyword **:DISPLAY** is included in *OPTIONS*, **ED** uses menus for any prompting, (e.g., to choose one of several possible definitions to edit). If **:DISPLAY** is not included, **ED** prints its queries to and reads the user's replies from ***QUERY-IO*** (usually the Exec in which you are typing). Thus all of the following are correct ways to call the editor:

```
(ED 'NAME :DISPLAY)
(ED 'NAME 'FUNCTIONS)
(ED 'NAME '(:DISPLAY))
(ED 'NAME '(FUNCTIONS :DISPLAY))
(ED 'NAME '(FUNCTIONS VARIABLES :DISPLAY))
```

The other keywords are interpreted as follows:

:CURRENT

This is a new option with Medley that causes ED to call TYPESOF with SOURCE=CURRENT. This prevents TYPESOF from searching FILECOMS and from looking in WHERE-IS databases. The **CURRENT** option looks only for definitions that are currently loaded. When you know that the definition is loaded, use of the **CURRENT** option results in ED being significantly faster.

:DONTWAIT

Lets the edit interface return right away, rather than waiting for the edit to be complete. **DF**, **DV**, **DC**, and **DP** specify this option now, so editing from the exec will not cause the exec to wait.

:NEW

Lets you install a new definition for the name to be edited. You will be asked what type of dummy definition you wish to install based on which file manager types were included in *OPTIONS*.

:COMPILE-ON-COMPLETION

This option specifies that the definition being edited should be compiled upon completion regardless of the completion command used.

:CLOSE-ON-COMPLETION

Tells the editor that it must close the editor window after the first completion. So in SEdit, CONTROL-X will close the window; shrinking the window is not allowed. Editor windows opened by the exec command **FIX** specify this option.

If *NAME* does not have a definition of any of the given types, **ED** can create a dummy definition of any of those types. If **:DISPLAY** is provided in *OPTIONS*, **ED** will pop-up the following menu asking you which type of definition to install. Select the template for the type of definition you wish to create from the DEFN menus and submenus:

```

Select a type for a dummy defn:
OPTIMIZERS
STRUCTURES
SETFS
TYPES
VARIABLES
FUNCTIONS
DEFINE-TYPES
FNS
Don't make a dummy defn

```

New kinds of dummy definitions can be added to the system through the use of the **:PROTOTYPE** option to **XCL:DEFDEFINER**.

Mapping the Old Edit Interface to ED

The old functions for starting the Lisp editor (**DF**, **DV**, **DP**, and **DC**) have been reimplemented so that they work with Common Lisp. The old edit commands map to the new editor function (ED) as follows:

```
DF NAME ⇒ (ED 'NAME '(FUNCTIONS FNS :DONTWAIT))
DV NAME ⇒ (ED 'NAME '(VARIABLES VARS :DONTWAIT))
DP NAME ⇒ (ED 'NAME '(PROPERTY-LIST :DONTWAIT))
DP NAME MYPROP ⇒ (ED '(NAME MYPROP) '(PROPS :DONTWAIT))
DC NAME ⇒ (ED 'NAME '(FILES :DONTWAIT))
```

Thus, for example, when **DF** is invoked it looks first for Common Lisp **FUNCTIONS** and then for Interlisp **FNS**. **DV**, **DP** and **DC** operate in an analogous fashion.

Editing Values Directly

The **TYPE** you specify for the object you want to edit determines how that object is edited, i.e. by **DEFINITION** or **VALUE**. Normally you want to edit the **DEFINITION** (this is the default). For example, suppose *FOO* is defined as a variable; to start the editor on the **DEFINITION** of *FOO*, use the form:

```
(ED 'FOO)    or  (ED 'FOO 'VARIABLES)
```

There may be times when you do not have access to the **DEFINITION** of an object that you need to edit. This can occur when you do not have the source code loaded. You can edit its **VALUE** directly using the form:

```
FOR VARIABLES: ⇒ (ED 'NAME 'IL:VARS)
```

```
FOR FUNCTIONS: ⇒ (ED 'NAME 'IL:FNS)
```

By starting the editor on the **VALUE** of an object, you can change its value without changing its definition. (AR 8971)

To start the editor on the **VALUE** of *FOO*, for example, use the form:

```
(ED 'FOO 'VARS)
```

EXAMPLE:

When you load a compiled file, the **DEFINITION** of an object is not loaded. Only the **VALUE** is loaded. The compiler does not store the defining forms for objects. Suppose you have compiled code for a system file loaded, but you do not have access to the sources that contain the **DEFINITIONS**, and you need to change the value of a system variable, say **NETWORKLOGINFO**. This variable has a defining form and the system knows this, but the form is not loaded and is not available. You can edit the **VALUE** of the variable directly using:

```
(ED 'NETWORKLOGINFO 'IL:VARS)
```

An editor window opens displaying the **VALUE** of **NETWORKLOGINFO**:

```

Sedit NETWORKLOGINFO Package: INTERLISP
((TENEX (LOGIN "LOGIN " USERNAME " " PASSWORD " ↑M")
  (ATTACH "ATTACH " USERNAME " " PASSWORD " ↑M")
  (WHERE "WHERE " USERNAME CR
    "ATTACH " USERNAME
    " " PASSWORD CR)))
(TOPS20 (LOGIN "LOGIN " USERNAME CR PASSWORD CR)
  (ATTACH "ATTACH " USERNAME "lama " CR PASSWORD CR)
  (WHERE "LOGIN " USERNAME CR PASSWORD CR))
(UNIX (LOGIN WAIT CR WAIT USERNAME CR WAIT PASSWORD CR))
(IFS (LOGIN "Login " USERNAME " " PASSWORD CR) (ATTACH))
(NS (LOGIN "Logon" CR USERNAME CR PASSWORD CR))
(VMS (LOGIN USERNAME CR PASSWORD CR)))

```

Section 16.18 Editor Functions

(II:16.74)

The function **FINDCALLERS** has the following limitations in Lisp:

1. **FINDCALLERS** only identifies by name the occurrences inside of Interlisp FNS, not Common Lisp FUNCTIONS.
2. Because **FINDCALLERS** uses a textual search, it may report more occurrences of the specified *ATOMS* than there actually are, if the file contains symbols by the same name in another package, or symbols with the same p-name but different alphabetic case. **EDITCALLERS** still edits only the actual occurrences, since it reads the functions and operates on the real Lisp structure, not its printed representation.

Chapter 17 File Package

The Interlisp-D File Package has been renamed the File Manager. Its operation is unchanged; however, it has been extended to manipulate, load and save Common Lisp functions, variables, etc. It also allows specification of the reader environment (package and readtable) to use when writing and reading a file, solving the problem of compatibility between old and new (Common Lisp) syntax.

Note that although source files from earlier releases can be loaded into Lyric, files produced by the File Manager in the Lyric release cannot be loaded into previous releases. This is true for several reasons, the most important being that previous releases did not have packages, so symbols cannot be read back consistently.

The new File Manager includes several new types to deal with the various definition forms supported in Xerox Common Lisp. The following table associates each new type with the forms that produce definitions of that type:

FUNCTIONS	CL:DEFUN, CL:DEFMACRO, CL:DEFINE-MODIFY-MACRO, XCL:DEFINLINE, XCL:DEFDEFINER, XCL:DEFINE-PROCEED-FUNCTION.
VARIABLES	CL:DEFCONSTANT, CL:DEFVAR, CL:DEFPARAMETER, XCL:DEFGLOBALVAR, XCL:DEFGLOBALPARAMETER
STRUCTURES	CL:DEFSTRUCT, XCL:DEFINE-CONDITION
TYPES	CL:DEFTYPE
SETFS	CL:DEFSETF, CL:DEFINE-SETF-METHOD
DEFINE-TYPES	XCL:DEF-DEFINE-TYPE
OPTIMIZERS	XCL:DEFOPTIMIZER
COMMANDS	XCL:DEFCOMMAND

Note that the types listed above, as well as all the old File Manager types, are symbols in the INTERLISP package. In addition, the "filecoms" variable of a file and its rootname are also both in the INTERLISP package. You should be careful when typing to a Common Lisp exec to qualify all such symbols with the prefix **IL:**; e.g.,

3>**(setq il:foocoms '((il:functions bar) (il:prop il:filetype il:foo)))**

to indicate you want the function BAR (in the current package) to live on a file with rootname FOO, and also that FOO's FILETYPE property should be saved.

Reader Environments and the File Manager

(II:17.1)

In order for **READ** to correctly read back the same expression that **PRINT** printed, it is necessary that both operations be performed in the same reader environment, i.e., the collection of parameters that affect the way the reader interprets the characters appearing on the input stream. In previous releases of Interlisp there was, for all practical purposes, a single such environment, defined entirely by the readtable *FILERDTBL*. In the Lyric release of Lisp there are two significantly different readtables in which to read (Common Lisp and Interlisp). In addition, there are more parameters than just the readtable that can potentially affect **READ**: the current package and the read base (the bindings of ***PACKAGE*** and ***READ-BASE***).

To handle this diversity, a new type of object is introduced, the **READER-ENVIRONMENT**, consisting of a readtable, a package, and a read/print base. Every file produced by the File Manager has a header at the beginning specifying the reader environment for that file. **MAKEFILE** and the compiler produce this header, while **LOAD**, **LOADFNS**, and other file-reading functions read the header in order to set their reading environment correctly. Files written in older releases of Lisp lack this header and are interpreted as

having been written in the environment consisting of the readtable *FILERDTBL* and the package *INTERLISP*. Thus, you need take no special action to be able to load Koto source files into Lyric; characters that are "special" in Common Lisp, such as colon, semi-colon and hash, are interpreted as the "ordinary" characters they were in Koto.

The File Manager's reader environments are specified as a property list of alternating keywords and values of the form **(:READTABLE *readtable* :PACKAGE *package* :BASE *base*)**. The **:BASE** pair is optional and defaults to 10. The values for *readtable* and *package* should either be strings naming a readtable and package, or expressions that can be evaluated to produce a readtable and package. In the former case, the readtable or package *must* be one that already exists in a virgin Lisp sysout (or at least in any Lisp image in which you might attempt *any* operation that reads the file). If an expression is used, care should be exercised that the expression can be evaluated in an environment where no packages or readtables, other than the documented ones, are presumed to exist. For hints and guidelines on writing the *package* expression for files that create or use their own private packages, please see Chapter 11 of the *Common Lisp Implementation Notes*.

When **MAKEFILE** is writing a source file, it uses the following algorithm to determine the reading environment for the new file:

1. If the root name for the file has the property **MAKEFILE-ENVIRONMENT**, the property's value is used. It should be in the form described above. Note that if you want the file always to be written in this environment, you should save the **MAKEFILE-ENVIRONMENT** property itself on the file, using a **(PROP MAKEFILE-ENVIRONMENT *file*)** command in the filecoms.
2. If a previous version of the file exists, **MAKEFILE** uses the previous version's environment. **MAKEFILE** does this even when given option **NEW** or the previous version is no longer accessible, assuming it still has the previous version's environment in its cache. If the previous version was written in an older release, and hence has no explicit reader environment, **MAKEFILE** uses the environment **(:READTABLE "INTERLISP" :PACKAGE "INTERLISP" :BASE 10)**.
3. If no previous version exists (this is a new file), **MAKEFILE** uses the value of ***DEFAULT-MAKEFILE-ENVIRONMENT***, initially **(:READTABLE "XCL" :PACKAGE "INTERLISP" :BASE 10)**.

Note that changing the value of ***DEFAULT-MAKEFILE-ENVIRONMENT*** only affects new files. If you decide you don't like the environment in which an existing file is written, you must give the file a **MAKEFILE-ENVIRONMENT** property to override any prior default.

Since the XCL readtable is case-insensitive, you should avoid using it for files that contain many mixed-case symbols or old-style Interlisp comments, as these will be printed with many escape

delimiters. This is why the default for reprinted Koto sources is the INTERLISP readtable.

The readtable named LISP (the pure Common Lisp readtable) should ordinarily not be used as part of a **MAKEFILE** environment. It exists solely for the use of "pure" Common Lisp (as in the CL Exec), and thus has no provision for font escapes (inserted by the Lisp prettyprinter) to be treated as whitespace. Most users will want to use either XCL or INTERLISP as the readtable for files.

If the environment for the new version of the file differs from that of the previous version, **MAKEFILE** copies unchanged FNS definitions by actually reading from the old file, rather than just copying characters as it otherwise would. Similarly, when **RECOMPILE** or **BRECOMPILE** attempt to recompile a file for which the previous compiled version's reader environment is different, they must compile afresh all the functions on the file, i.e., they behave like **TCOMPL** or **BCOMPL**.

Modifying Standard Readtables

In the past, programmers have been periodically tempted to change standard readtables, such as **T** and **FILERDTBL**, typically by adding macros to read certain objects in a convenient way. For example, the PQUOTE LispUsers module defined single quote as a macro in **FILERDTBL**. Unfortunately, changing a standard readtable means that unless you are very careful, you cannot read other users' files that were not written with your change, and they cannot read your files without obtaining your macro. Furthermore, the effects are often subtle. Rather than breaking, the system merely reads the file incorrectly. For example, reading a file written with PQUOTE in an environment lacking PQUOTE produces many symbols with a single quote packed on the front.

This confusion can be avoided with **MAKEFILE** reader environments. To add your own special macro:

1. Copy some standard readtable; e.g., (COPYRDTBL "INTERLISP").
2. Give it a distinguished name of its own, by using (READTABLEPROP *rdtbl* 'NAME "yourname").
3. Make your change in the copied readtable.
4. Use your new private readtable to write your files: use its name ("yourname") in the **MAKEFILE-ENVIRONMENT** property of selected files and/or change ***DEFAULT-MAKEFILE-ENVIRONMENT*** to affect all your new files.
5. Make sure to save your new readtable. It is usually most convenient to include the code to create it (steps 1-3) in your system initialization, but you could even write a self-contained expression to use in a single file's **MAKEFILE-ENVIRONMENT** property.

With this strategy, your system will read all files in the proper environment—your own files with your private readtable and other users' files in their environments, including the standard environments, which you have carefully avoided polluting. If

another user tries to load one of your files into an environment that doesn't know about your private readtable, **LOAD** will give an error immediately (readtable not found), rather than loading the file quietly but incorrectly.

Programmer's Interface to Reader Environments

The following function and macro are available for programmers to use. Note that reader environments only control the parameters that determine read/print consistency. There are other parameters, such as ***PRINT-CASE***, that affect the appearance of the output without affecting its ability to be read. Thus, reader environments are not sufficient to handle problems of, for example, repainting expressions on the display in exactly the same total environment in which they were first written.

(**MAKE-READER-ENVIRONMENT** *PACKAGE READTABLE BASE*) [Function]

Creates a **READER-ENVIRONMENT** object with the indicated components. The arguments must be valid values for the variables ***PACKAGE***, ***READTABLE*** and ***PRINT-BASE***; names are not sufficient. If any of the arguments is **NIL**, the current value of the corresponding variable is used. Thus (**MAKE-READER-ENVIRONMENT**) returns an object that captures the current environment.

(**WITH-READER-ENVIRONMENT** *ENVIRONMENT . FORMS*) [Macro]

Evaluates each of the *FORMS* with ***PACKAGE***, ***READTABLE***, ***PRINT-BASE*** and ***READ-BASE*** bound to the values in the *ENVIRONMENT* object. Both ***PRINT-BASE*** and ***READ-BASE*** are bound to the single *BASE* value in the environment.

(**GET-ENVIRONMENT-AND-FILEMAP** *STREAM DONTCACHE*) [Function]

Parses the header of a file produced by the File Manager and returns up to four values:

1. The reader environment in which the file was written;
2. The file's "filemap", used to locate functions on the file;
3. The file position where the **FILECREATED** expression starts;
and
4. A value used internally by the File Manager.

STREAM can be a full file name, in which case this function returns **NIL** unless the information was previously cached. Otherwise, *STREAM* is a stream open for input on the file. It must be randomly accessible (unless information is available from the cache). If the file is in Common Lisp format (it begins with a comment), then value 1 is the default Common Lisp reader environment (readtable **LISP**, package **USER**) and the other values are **NIL**. Otherwise, if the file is not in File Manager format, values 1 and 2 are **NIL**, 3 is zero.

If *DONTCACHE* is true, the function does not cache any information it learns about File Manager files; otherwise, the information is cached to speed up future inquiries.

Section 17.1 Loading Files

(II:17.5)

Integration of Interlisp and Common Lisp **LOAD** functions

There are four kinds of files that can be loaded in Lisp:

1. Interlisp and Common Lisp source files produced by the File Manager using, for example, the **MAKEFILE** function.
2. Standard Common Lisp source files produced with a text editor either in Lisp or from some other Common Lisp implementation.
3. DFASL files of compiled code, produced by the new XCL Compiler, **CL:COMPILE-FILE** (extension DFASL)
4. LCOM files of compiled code, produced by the old Interlisp Compiler (**BCOMPL**, **TCOMPL**).

Types 1 and 4 were the only kind of files that you could load in Koto; types 2 and 3 are new with Lyric. Both **IL:LOAD** and **CL:LOAD** are capable of loading all four kinds of files. However, they use the following rules to make the types of files unambiguous so that they can be loaded in the correct reader environment.

- If the file begins with an open parenthesis (possibly after whitespace and font switch characters), it is assumed to be of type 1 or 4: files produced by the File Manager. The first expression on the file (at least) is assumed to be written in the old **FILERDTBL** environment; for new Lyric files this expression defines the reader environment for the remainder of the file. See the section, Reader Environments and File Manager for details.
- If the file begins with the special FASL signature byte (octal 221), it is assumed to be a compiled file in FASL format, and is processed by the FASL loader. The FASL loader ignores the **LDFLG** argument to **IL:LOAD**, treating all files as though **LDFLG** were **SYSLOAD** (redefinition occurs, is not undoable, and no File Manager information is saved).
- If the file begins with a semicolon, it is assumed to be a pure Common Lisp file. The expressions on the file are read with the standard Common Lisp readtable and in package **USER** (unless a package argument was given to **LOAD**; see below).
- If the file begins with any other character, **LOAD** doesn't know what to do. Currently, it treats the file as a pure Common Lisp file (as if it started with a comment).

Thus, if you prepare Common Lisp text files you should be sure to begin them with a comment so that **LOAD** can tell the file is in Common Lisp syntax.

The function **CL:LOAD** accepts an additional keyword **:PACKAGE**, whose value must be a package object; the function **IL:LOAD** similarly has an optional fourth argument **PACKAGE**. If a package argument is given, then **LOAD** reads Common Lisp text files (type 2 above) with ***PACKAGE*** bound to the specified package. In the

case of File Manager files (types 1 and 4), the package argument overrides the package specified in the file's reader environment.

(II:17.6-17.8)

The Interlisp functions **LOADFNS**, **LOADFROM**, **LOADVARS** and **LOADCOMP** do not work on FASL files. They do still work on files produced by the old compiler (extension LCOM).

(II:17.9)

FILESLOAD (also used by the File Manager's **FILES** command) now searches for compiled files by looking for a file by the specified name whose extension is in the list ***COMPILED-EXTENSIONS***. The default value for ***COMPILED-EXTENSIONS*** in the Lyric release is (DFASL LCOM). It searches the list of extensions in order for each directory on the search path. This means that FASL files are loaded in preference to old-style compiled files.

Section 17.2 Storing Files

The Lyric release contains two different compilers, the Interlisp Compiler that was present in Koto and previous releases, and the new XCL Compiler (see the next section, Chapter 18 Compiler). With more than one compiler available, the question arises as to which compiler will be used by the functions **CLEANUP** and **MAKEFILE**. The default behavior of these functions in Lyric is to always use the new XCL Compiler. This default can be changed, either on a file-by-file basis or system-wide. Most users, however, will have no need to change the default.

When the **C** or **RC** option has been given to **MAKEFILE**, the system first looks for the value of the **FILETYPE** property on the symbol naming the file. For example, for the file "{DSK}<LISPPFILES>MYFILE", the property list of the symbol **MYFILE** would be examined.

The **FILETYPE** property should be either a symbol from the list below or a list containing one of those symbols. The following symbols are allowed and have the given meanings:

- | | |
|----------------------|---|
| :TCOMPL | Compile this file by calling either TCOMPL or RECOMPILE , depending upon which of the C or RC options was passed to MAKEFILE . |
| :BCOMPL | Compile this file by calling either BCOMPL or BRECOMPILE , depending upon which of the C or RC options was passed to MAKEFILE . This is equivalent to the Koto behavior. |
| :COMPILE-FILE | Compile this file by calling CL:COMPILE-FILE , regardless of which option was passed to MAKEFILE . |

If no **FILETYPE** property is found, then the function whose name is the value of the variable ***DEFAULT-CLEANUP-COMPILER*** is used. The only legal values for this variable are **TCOMPL**, **BCOMPL**, and **CL:COMPILE-FILE**. Initially, ***DEFAULT-CLEANUP-COMPILER*** is set to **CL:COMPILE-FILE**.

If you choose to set the **FILETYPE** property of file name, you should take care that the filecoms for that file saves the value of that property on the file. This will ensure that the same compiler

will be used every time the file is loaded. To save the value of the property, you should include a line in the coms like the following:

```
(PROP FILETYPE MYFILE)
```

where MYFILE is the symbol naming your file.

Section 17.8.2 Defining New File Manager Types

(II:17.30)

The File Manager has been extended to allow File Manager types that accept any Lisp object as a name. A consequence of this is that any user-defined type's **HASDEF** function should be prepared to accept objects other than symbols as the *NAME* argument. Names are compared using **EQUAL**.

Definers: A New Facility for Extending the File Manager

The Definer facility is provided to make the process of adding a certain common kind of File Manager type easy. All of the new File Manager types in the Lyric release (including **FUNCTIONS**, **VARIABLES**, **STRUCTURES**, etc.) and almost all of the new defining macros (including **CL:DEFUN**, **CL:DEFPARAMETER**, **CL:DEFSTRUCT**, etc.) were themselves created using the Definer facility.

In previous releases, adding new types and commands to the File Manager involved deeply understanding the way in which it worked and defining a number of functions to carry out certain operations on the new type/command. Further, making functions and macros save away definitions of the new type was similarly subtle and generally difficult or complicated to do. With the addition of Common Lisp, it was realized that a large number of new types and commands would be added, all needing essentially the same implementation of the various operations. In addition, many new defining macros were to be added and all of them needed to save definitions.

As an explanation of the Definer facility, we will describe how **VARIABLES** and **CL:DEFPARAMETER** could be added into the system, if they were not already there.

First, a little background about our example. The macro **CL:DEFPARAMETER** is used in Common Lisp to globally declare a given variable to be special and to give it an initial value. (For the purposes of this example, we will ignore the documentation-string given to real **CL:DEFPARAMETER** forms.) The value of a call to the macro should be the name of the variable being defined. An acceptable definition of this macro might appear as follows:

```
(DEFMACRO CL:DEFPARAMETER (SYMBOL EXPRESSION)
  `(PROGN
    (CL:PROCLAIM '(CL:SPECIAL ,SYMBOL))
    (SETQ ,SYMBOL ,EXPRESSION)
    ',SYMBOL))
```

There are some problems with using such a simple definition in the Lisp environment, however. For example, if a call to this macro were typed to the Exec, the File Manager would not be told to notice it. Thus, there would be no convenient way to remember to

add the form to the filecoms of some file and thus to save it away. Also, note that the macro does not pay attention to the **DFNFLG** variable; thus, loading a file containing a **CL:DEFPARAMETER** form would always set the variable to the value of the initial expression, even when **DFNFLG** was set to **ALLPROP**. This could make editing code using this variable difficult.

We will now proceed to fix these problems by getting the Definer facility involved. There are two steps involved in using Definers:

- Unless one of the currently-existing File Manager types is appropriate for definitions using the new macro, a new type must be created. The macro **XCL:DEF-DEFINE-TYPE** is used for this purpose.
- The macro must be defined in such a way that the File Manager can tell that it should notice and save uses of the macro and under which File Manager type the uses should be saved. The macro **XCL:DEFDEFINER** is used for this purpose.

Since we are pretending for the example that the File Manager type **VARIABLES** is not defined, we decide that definitions using **CL:DEFPARAMETER** should not be given any of the already-existing types. We must define a type, therefore, and we decide to call it **VARIABLES**. The following **XCL:DEF-DEFINE-TYPE** form will do the trick:

```
(XCL:DEF-DEFINE-TYPE VARIABLES "Common Lisp
variables")
```

The first argument to **XCL:DEF-DEFINE-TYPE** is the name for the new type. The second argument is a descriptive string, to be used when printing out messages about the type.

With the new type thus created, we can now use **XCL:DEFDEFINER** to redefine the macro. Simply changing the word **DEFMACRO** into **XCL:DEFDEFINER** and adding an argument specifying the new type suffices to change our earlier definition into a use of the Definer facility:

```
(XCL:DEFDEFINER CL:DEFPARAMETER VARIABLES
                (SYMBOL EXPRESSION)
  `(PROGN
    (CL:PROCLAIM '(CL:SPECIAL ,SYMBOL))
    (SETQ ,SYMBOL ,EXPRESSION)
    ',SYMBOL))
```

(In fact, we could also remove the final **',SYMBOL**; **XCL:DEFDEFINER** automatically arranges for the new macro to return the name of the new definition.) Now, if we were to type the form

```
(CL:DEFPARAMETER *FOO* 17)
```

into the Exec and then call the function **FILES?**, we would be presented with something like the following:

```
24> (FILES?)
the Common Lisp variables: *FOO*
...to be dumped. want to say where the above go?
```

As with other File Manager types, our definitions are being kept track of. If we answer Yes to the above question and specify a file

in which to save the definition, a command like the following will be added to the filecoms:

```
(VARIABLES *FOO*)
```

Actually, the output from **FILES?** as shown above is not quite accurate. In reality, we would also be asked about the following:

```
the Common Lisp functions/macros: CL:DEFPARAMETER
the Definition types: VARIABLES
```

The File Manager is also watching for new types and new Definers being created and will let us save those definitions as well. These would be listed in the filecoms as follows:

```
(DEFINE-TYPES VARIABLES)
(FUNCTIONS CL:DEFPARAMETER)
```

All of these definitions are full-fledged File Manager citizens. The functions **GETDEF**, **HASDEF**, **PUTDEF**, **DELDEF**, etc. all work with the new type. We can edit the definition of ***FOO*** above simply by specifying the type to the **ED** function:

```
(ED '*FOO*' 'VARIABLES)
```

When we exit the editor, the new definition will be saved and, unless **DNFLG** is set to **PROP** or **ALLPROP**, evaluated.

It is now time to fully describe the macros **XCL:DEF-DEFINE-TYPE** and **XCL:DEFDEFINER**.

XCL:DEF-DEFINE-TYPE *NAME DESCRIPTION &KEY :UNDEFINER* [Macro]

Creates a new File Manager type and command with the given *NAME*. The string *DESCRIPTION* will be used to describe the type in printed messages. The new type implements **PUTDEF** operations by evaluating the definition form, **GETDEF** and **HASDEF** by looking up the given name in an internal hash-table, using **EQUAL** as the equality test on names, and **DELDEF** by removing any named definition from the hash-table. If the **:UNDEFINER** argument is provided, it should be the name of a function to be called with the *NAME* argument to any **DELDEF** operations on this type. The **:UNDEFINER** function can perform any other operations necessary to completely delete a definition.

XCL:DEF-DEFINE-TYPE forms are File Manager definitions of type **DEFINE-TYPES**.

As an example of the full use of **XCL:DEF-DEFINE-TYPE**, here is the complete definition of the type **VARIABLES** as it exists in the Lyric release:

```
(XCL:DEF-DEFINE-TYPE VARIABLES "Common Lisp variables"
:UNDEFINER UNDOABLY-MAKUNBOUND)
```

The function **UNDOABLY-MAKUNBOUND** is described in Appendix D of these Release Notes.

XCL:DEFDEFINER *{NAME} (NAME {OPTION}*) TYPE ARG-LIST &BODY BODY* [Macro]

Creates a macro named *NAME*, calls to which are seen as File Manager definitions of type *TYPE*. *TYPE* must be a File Manager type previously defined using **XCL:DEF-DEFINE-TYPE**. *ARG-LIST* and *BODY* are precisely as in **DEFMACRO**. A macro defined using

XCL:DEFDEFINER differs from one defined using **DEFMACRO** in the following ways:

- *BODY* will be evaluated if and only if the value of **DFNFLG** is not one of **PROP** or **ALLPROP**.
- The form returned by *BODY* will be evaluated in a context in which the File Manager has been temporarily disabled. This allows Definers to expand into other Definers without the subordinate ones being noticed by the File Manager.
- Calls to Definers return the name of the new definition (as, for example, **CL:DEFUN** and **CL:DEFPARAMETER** are defined to do).
- Calls to Definers are noticed and remembered by the File Manager, saved as a definition of type *TYPE*.
- SEdit- and Interlisp-style comment forms (those with a CAR of IL:*) are stripped from the macro call before it is passed to *BODY*. (This comment-removal is partially controlled by the value of the variable ***REMOVE-INTERLISP-COMMENTS***, described below.)

The following *OPTIONS* are allowed:

(:UNDEFINER *FN*)

If **DELDEF** is called on a name whose definition is a call to this Definer, *FN* will be called with one argument, the name of the definition. This option allows for Definer-specific actions to be taken at **DELDEF** time. This is useful when more than one Definer exists for a given type. *FN* should be a form acceptable as the argument to the **FUNCTION** special form.

(:NAME *NAME-FN*)

By default, the Definer facility assumes that the first argument to any macro defined using **XCL:DEFDEFINER** will be the name under which the definition should be saved. This assumption holds true for almost all Common Lisp defining macros, including **CL:DEFUN**, **CL:DEFMACRO**, **CL:DEFPARAMETER** and **CL:DEFVAR**. It doesn't work, however, for a few other forms, such as **CL:DEFSTRUCT** and **XCL:DEFDEFINER** itself. When defining a macro for which that assumption is false, the **:NAME** option should be used. *NAME-FN* should be a function of one argument, a call to the Definer. It should return the Lisp object naming the given definition (most commonly a symbol, but any Lisp object is permissible). For example, the **:NAME** option in the definitions of **CL:DEFSTRUCT** and **XCL:DEFDEFINER** is as follows:

```
( :NAME (LAMBDA (FORM)
      (LET ((NAME (CADR FORM)))
        (COND ((LITATOM NAME)
              NAME)
              (T (CAR NAME))))))
```

NAME-FN should be a form acceptable as the argument to the **FUNCTION** special form (i.e., a symbol naming a function or a LAMBDA-form).


```
( :PROTOTYPE DEFN-FN)
```

When the editor function **ED** is passed a name with no definitions, the user is offered a choice of several ways to create a prototype definition. Those choices are specified with the **:PROTOTYPE** option to **XCL:DEFDEFINER**. *DEFN-FN* should be a function of one argument, the name to be defined using this Definer. *DEFN-FN* should return either NIL, if no definition of that name can be created with this Definer, or a form that, when evaluated, would create a definition of that name. For example, the **:PROTOTYPE** option for **CL:DEFPARAMETER** might look as follows:

```
( :PROTOTYPE (LAMBDA (NAME)
              (AND (LITATOM NAME)
                   `(CL:DEFPARAMETER ,NAME "Value"))))
```

An example using all of the features of **XCL:DEFDEFINER** is the definition of **XCL:DEFDEFINER** itself, which begins as follows:

```
(XCL:DEFDEFINER (XCL:DEFDEFINER
                 (:UNDEFINER \DELETE-DEFINER)
                 (:NAME
                  (LAMBDA (FORM)
                    (LET ((NAME (CADR FORM)))
                      (COND ((LITATOM NAME)
                           NAME)
                           (T (CAR NAME)))))))
                 (:PROTOTYPE
                  (LAMBDA (NAME)
                    (AND (LITATOM NAME)
                        `(XCL:DEFDEFINER ,NAME "Type"
                                           ("Arg List"
                                            "Body")))))
                 FUNCTIONS
                 (NAME-AND-OPTIONS TYPE ARG-LIST &BODY BODY)
                 ...)
```

The following variable is used in the process of removing SEdit- and Interlisp-style comments from Definer forms:

REMOVE-INTERLISP-COMMENTS [Variable]

Interlisp-style comments are forms whose **CAR** is the symbol **IL:***. It is possible for certain lists in Lisp code to begin with **IL:*** but not be a comment (for example, a **SELECTQ** clause). When such a list is discovered, the value of ***REMOVE-INTERLISP-COMMENTS*** is examined. If it is **T**, the list is assumed to be a comment and is removed without comment. If it is **:WARN**, a warning message is printed, saying that a possible comment was not stripped from the code. If ***REMOVE-INTERLISP-COMMENTS*** is **NIL**, the list is not removed, but no warning is printed. This variable is initially set to **:WARN**.

(CL:EVAL-WHEN WHEN COM₁ ... COM_N) [File Package Command]

Interprets each of the commands *COM₁ ... COM_N* as a file package command, but output is wrapped in **CL:EVAL-WHEN**.

EXAMPLE:

```
(CL:EVAL-WHEN (CL:EVAL CL:COMPILE)
  (OPTIMIZERS FOO))
```

will cause the following to be written to the file:

```
(CL:EVAL-WHEN (CL:COMPILE)
 (DEFOPTIMIZER FOO <optimizer for FOO>))
```

Chapter 18 Compiler

The Lyric release contains two distinct Lisp compilers:

- The Interlisp Compiler, described in detail in Section 18 of the *IRM*,
- The new XCL Compiler, described in the *Common Lisp Implementation Notes*.

The Interlisp Compiler provides compatibility with previous releases of Interlisp-D. It continues to work in very much the same way as it did in Koto; as before, it compiles all of the Interlisp language. The Interlisp Compiler does not, however, compile the Common Lisp language and will not be extended to do so. The Lyric release is the last release to contain the Interlisp Compiler as a component; future releases will have only the new XCL Compiler. The XCL Compiler is designed to handle both Interlisp and Common Lisp.

Several incompatible changes have been made in the compiled object code produced by the Interlisp Compiler. This means that *all user code must be recompiled in Lyric*. Code compiled in Koto or previous releases will not load into Lyric, and code compiled in Lyric will not load into earlier releases. The filename extension for Interlisp compiled files has been changed from DCOM to LCOM in order to minimize possible confusion.

The XCL Compiler writes its output on a new kind of object file, the DFASL file. These files are quite different from the DCOM/LCOM files produced by the Interlisp Compiler. DFASL files are somewhat more compact, much faster to load and can represent a wider range of data objects than was possible in LCOMs.

Interlisp source files from Koto can be compiled using the new XCL compiler. However, some files need to be remade in Lyric before compilation: files containing bitmaps, Interlisp arrays, or the **UGLYVARS** and/or **HORRIBLEVARS** File Manager commands. To compile such a file, first **LOAD** it, then call **MAKEFILE** to write it back out. This action causes the bitmaps and other unusual objects to be written back in a format acceptable to the new compiler.

The default behavior of the File Manager's **CLEANUP** and **MAKEFILE** functions is to use the new XCL Compiler to compile files, rather than the old Interlisp Compiler. To change this behavior, see Section 17.2, Storing Files.

Note that if you call the compiler explicitly, rather than via **CLEANUP** or **MAKEFILE**, you should be careful to specify the correct compiler. The new compiler is invoked by calling **CL:COMPILE-FILE**. If you inadvertently call **BCOMPL** on a file for which **CLEANUP** has routinely been using the new XCL compiler, there are two undesirable consequences: (1) Any Common Lisp

functions on the file will not be compiled (the Interlisp compiler does not recognize **CL:DEFUN**), and (2) the DFASL files produced by earlier calls on the XCL compiler will still be loaded by **FILESLOAD** in preference to the LCOM file produced by **BCOMPL**.

Lisp provides a facility, **XCL:DEFOPTIMIZER**, by which you can advise the compiler about efficient compilation of certain functions and macros. **XCL:DEFOPTIMIZER** works with both the old Interlisp Compiler and the Lyric XCL Compiler. See the *Common Lisp Implementation Notes* for a description of the compiler.

Warning when Loading Compiled Files

CAUTION: Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by either the XCL compiler or SEdit. It is possible to insert a comment at the beginning of your function that looks like

(* DECLARATIONS: --)

The tail, or -- section, of this comment is taken as a set of local record declarations which are then used by the compiler in that function just as if they had been declared globally. The XCL compiler does not directly support this feature. If the body of the function gets DWIMIFIED for some other reason, the record declarations will happen to be noticed, otherwise they will not be seen and the compiler will signal an error if it can't find an appropriate top-level record definition.

There are two caveats that you should note:

1. The compiler will give error messages "undefined record name ..." for the records that are declared this way, but will generate correct code.
2. SEdit does not recognize such declarations. Thus, if the "Expand" command is used in SEdit, the expansion will not be done with these record declarations in effect. The code that you see in the editor will not be the same code compiled by the BYTECOMPILER.

Section 18.3 Local Variables and Special Variables

(II:18.5)

The new execs always use the Common Lisp interpreter, causing LET and PROG statements at top level, particularly in a so-called Interlisp exec, to create lexical bindings, rather than deep or "special" bindings. This can be worked around by setting **il:specvars** to T, which will cause the interpreter to create special bindings for all variables. This can also be worked around by wrapping the form to be "interlisp evaluated" in the IL:INTERLISP special form, which causes the Interlisp interpreter to be invoked.

Chapter 19 Masterscope

Masterscope is now a Lisp Library Module, not part of the environment.

Chapter 21 CLISP

CLISP infix forms do not work under the Common Lisp evaluator; only "clean" CLISP prefix forms are supported. You should run DWIMIFY in Koto on all other CLISP code before attempting to load it in Lyric. The remainder of this note describes the specific limitations on CLISP in Lyric.

There are two broad classes of transformations that DWIM applies to Lisp code:

1. A sort of macro expander that transforms **IF**, **FOR**, **FETCH**, etc. forms into "pure" Lisp code in well-defined ways.
2. A heuristic "corrector" that performs spelling correction and transforms CLISP infix forms such as X+Y into (PLUS X Y), sometimes having to make guesses as to whether X+Y might really have been the name of a variable.

An operational way of distinguishing the two is that DWIMIFY applied to code of type (1) makes no alterations in the code, whereas for code of type (2) it physically changes the form. Another difference is that code of type (2) must be dwimified before it can be compiled (user typically sets **DWIMIFYCOMPFLG** to T), whereas the compiler is able to treat code of type (1) as a special kind of macro.

Broadly speaking, code of type (2) is no longer fully supported. In particular, DWIM is invoked only when the code is encountered by the Interlisp evaluator. This means code typed to an "Old Interlisp" Executive, and code inside of an interpreted Interlisp function. Furthermore, some CLISP infix forms no longer DWIMIFY correctly. It is likely that CLISP infix will not be supported at all in future releases.

Expressions typed to the new Executives and inside of Common Lisp functions are run by the Common Lisp evaluator (**CL: EVAL**). As far as this evaluator is concerned, DWIM does not exist, and forms beginning with "CLISP" words (**IF**, **FOR**, **FETCH**, etc) are macros. These macros perform no DWIM corrections, so all of the subforms must be correct to begin with. This is a change from past releases, where the DWIM expansion of a CLISP word form also had the side effect of transforming any CLISP infix that it might have contained. For example, the macro expansion of

```
(if X then Y+1)
```

treats Y+1 as a variable, rather than as an addition. The correct form is

```
(if X then (PLUS Y 1)),
```

which is the way an explicit call to DWIMIFY would transform it.

If you have CLISP code from Koto you are advised to DWIMIFY the code before attempting to run or compile it in Lyric. Because of differences in the environments, not all CLISP constructs will DWIMIFY correctly in Lyric. In particular, the following do not work reliably, or at all:

1. The list-composing constructs using < and > do not DWIMIFY if the < is unpacked (an isolated symbol), because in Common Lisp, < is a perfectly valid CAR of form. On the other hand, the closing > *must* be unpacked if the last list element is quoted, since, for example, (<A 'B>) reads as (<A (QUOTE B>)).
2. Because of the conventional use of the characters * and - in Common Lisp names, those characters are only recognized as CLISP operators when they appear unpacked.
3. On the other hand, the operators + and / are the names of special variables in Common Lisp (Steele, p. 325), and hence cause no error when passed unpacked to the evaluator. Thus (LIST X + Y) returns a list of three elements, with no resort to DWIM; however, the parenthesized version (LIST (X + Y)) and the packed version (LIST X+Y) both work.

If you routinely DWIMIFY code, so that no CLISP infix forms (type 2 above) remain on your source files, you may not need to make any changes. However, note that the fact that DWIMIFY of prefix forms implicitly performed infix transformations can hide code that escaped being completely dwimified before being written to a file.

There is a further caution regarding even routinely dwimified code that has not been edited since before Koto. Two uses of the assignment operator (_) no longer work, if not explicitly dwimified, because their canonical form (the output of DWIMIFY) has changed, and the old form is no longer supported when the form is simply evaluated, macro-expanded, or compiled (with **DWIMIFYCOMPFLG = NIL**):

1. Iterative statement bindings must always be lists. For example, the old form

```
(bind X_2 for Y in --)
```

is now canonically

```
(bind (X _ 2) for Y in --).
```

2. In a WITH expression, assignments must be dwimified to remove `_`. For example, the old form

```
(with MYRECORD MYFIELD _ (FOO))
```

is now canonically

```
(with MYRECORD (SETQ MYFIELD (FOO))).
```

DWIMIFY in Koto correctly made these transformations; however, in some older releases, it did not. Such old code must be explicitly dwimified (which you can do for these cases in Lyric). The errors resulting from failure to do so can be subtle. In particular, the compiler issues no special warning when such code is compiled. For example, in case 1, the macro expansion of the old form treats the symbol `x_2` as a variable to bind, rather than as a binding of the variable `x` with initial value 2. The only hint from the compiler that anything is amiss is likely to be an indication that the variable `x` is used freely but not bound. Case 2 is even subtler: the symbols `MYFIELD` and `_` are treated as symbols to be evaluated; since their values are not used, the compiler optimizes them away, reducing the entire expression to simply `(FOO)`, and there is thus no warning of any sort from the compiler.

Chapter 22 Performance Issues

Section 22.3 Performance Measuring

(II:22.8)

The Interlisp-D **TIME** function has been withdrawn and replaced with the Common Lisp **TIME** macro (the symbol **TIME** is shared between IL and CL and thus need not be typed with a package prefix). The functionality of the *TIMEN* and *TIMETYP* arguments to the old **TIME** can be had by keywords to the **TIME** macro. The *Common Lisp Implementation Notes* describe the new **TIME** macro and its associated command in more detail.

[This page intentionally left blank]

VOLUME III—INPUT/OUTPUT

Chapter 24 Streams and Files

The Xerox Common Lisp file system supports multiple streams open simultaneously on the same file. This is an *incompatible change* to the semantics of Interlisp-D. You may have to modify old programs if they have not followed the guidelines listed in Sec 24.5 of the *Interlisp-D Reference Manual*. Some of the implications of this change for Interlisp programs are described below.

In prior releases of Interlisp-D, the system treated the *name* of an open file as a synonym for the *stream* open on the file. This meant that only one stream could be open at any time on a given file. In the Lyric release, a file name is no longer a unique name for an open stream. Thus, file names are no longer acceptable as the file/stream argument to any input/output or file system function that operates on an open stream (**READ**, **PRINT**, **CLOSEF**, **COPYBYTES**, etc). The only non-stream values acceptable as stream designators are the symbols **NIL** and **T**, designating the primary and terminal input/output streams. An attempt to use a litatom, even a "full file name," as a stream designator signals the error "LITATOM 'streams' no longer supported." Strings no longer designate an input stream whose source is the string itself—programs should call **OPENSTRINGSTREAM** instead, or use the comparable Common Lisp forms, such as **CL:WITH-INPUT-FROM-STRING**.

The functions **OPENFILE** and **OPENSTREAM** are now synonymous—both return a stream instead of a "full file name." The functions **INPUT** and **OUTPUT** also return streams. One exception to this is that **INPUT** and **OUTPUT** return **T** in the case where the primary input or output stream was previously directed to the terminal. However, this special behavior is for the Lyric release only; we recommend that you convert old code that depended on testing (**EQ (OUTPUT) T**). Note that the values of the variables ***STANDARD-INPUT*** and ***STANDARD-OUTPUT*** are always streams, even if they are directed to the terminal.

The function **FULLNAME** can be used to obtain the name of a stream. For your convenience, the print syntax of streams now includes the name of the stream (if to a file) and its access (input, output, etc.). Functions, such as **UNPACKFILENAME**, that manipulate file names generally accept a stream as well, extracting the name of the file from the stream.

INFILEP still returns a full file name, as it is merely recognizing a file, not opening a stream to it. Programmers should be wary of code that subsequently tries to use the value of **INFILEP** as a stream argument. And, of course, the **FILENAME** argument to **OPENSTREAM** is still a name (a symbol or string), not a stream. **OPENSTREAM** also accepts a Common Lisp pathname as its **FILENAME** argument.

The function **CLOSEALL** is no longer implemented. The function **OPENP** returns **NIL** when passed a file name (or anything else but an open stream). However, for the Lyric release, (**OPENP NIL**) still returns a list of all streams open to files.

The functions **GETFILEINFO** and **SETFILEINFO** can still be given either an open stream or a file name. However, in the latter case, the request refers to the file, not to any stream open on the file. Thus, requesting the value of the attribute **LENGTH** for a file name may return a different value than requesting the value of the attribute **LENGTH** for a stream currently open on the file. **GETFILEINFO** returns **NIL** if given a file name and an attribute that only makes sense for streams (e.g., **ACCESS**, **ENDOFSTREAMOP**).

There is no difference between Common Lisp and Interlisp streams. A stream opened by an Interlisp function can be passed as argument to a Common Lisp input/output function, and vice versa.

Even though multiple streams per file are supported, the streams must still obey consistent access rules. That is, if a stream is open for output, no other streams on that file can be opened. It is not possible to **RENAMEFILE** or **DELFILE** a file that has *any* open stream on it.

The RS-232 or TTY ports are inherently single-user devices (rather than real files) thus, multiple streams cannot be open simultaneously on RS-232 or TTY.

Section 24.15 Deleting, Copying, and Renaming Files

(III:24.15)

The support of multiple streams per file now makes it possible to use **COPYFILE** without worrying about there being other readers of the file, in particular even when there is already a stream open on the file for sequential-only access (a case that failed in prior releases). Of course, **COPYFILE** cannot be used if the file already has an *output* stream open.

Chapter 25 Input/Output Functions

Variables Affecting Input/Output

There are several implicit parameters that affect the behavior of the input/output functions: the numeric print base, the primary output file, etc. In Common Lisp, these parameters are controlled by binding special variables. In Interlisp they are controlled by a functional interface; e.g., an output expression is wrapped in (**RESETFORM** (**RADIX** 8) --) to cause numbers to be printed in octal.

Where the input/output parameters in Common Lisp and Interlisp have essentially the same semantics, they have been integrated in

Lisp. That is, binding the Common Lisp special variable and calling the Interlisp function are equivalent operations, and they affect both Interlisp and Common Lisp input/output. However, it is considerably more efficient to bind a special variable than to call a function in a **RESETFORM** expression. In addition, binding a variable has only a local effect, whereas calling a function to side-effect the input/output parameters can also affect other processes. Thus, you are encouraged to use special variable binding to change parameters formerly changed via functional interface.

All of these variables are accessible in both the Common Lisp and Interlisp packages, so no package qualifier is required when typing them.

These parameters are as follows:

- | | |
|---|---|
| *PRINT-BASE* vs RADIX | Binding *PRINT-BASE* to an integer <i>n</i> from 2 to 36 tells the printing functions to print numbers in base <i>n</i> . This is equivalent to (RADIX <i>n</i>). Note: this variable should not be confused with *PRINT-RADIX* , another Common Lisp variable that controls whether Common Lisp functions include radix specifiers when printing numbers. |
| *STANDARD-INPUT* vs INPUT | Binding *STANDARD-INPUT* to an input stream specifies the stream from which to read when an input function's stream argument is NIL or omitted. Evaluating *STANDARD-INPUT* is the same as evaluating (INPUT), except that (INPUT) returns T if the primary input for the process is the same as the terminal input stream (this compatibility feature is for the Lyric release only). |
| *STANDARD-OUTPUT* vs OUTPUT | Binding *STANDARD-OUTPUT* to an output stream specifies the stream to which to print when an output function's stream argument is NIL or omitted. Evaluating *STANDARD-OUTPUT* is the same as evaluating (OUTPUT) except that (OUTPUT) returns T if the primary output for the process is the same as the terminal output stream (this compatibility feature is for the Lyric release only). |
| *PRINT-LEVEL* & *PRINT-LENGTH*
vs PRINTLEVEL | Binding *PRINT-LEVEL* to a positive integer <i>a</i> and *PRINT-LENGTH* to a positive integer <i>d</i> is equivalent to calling (PRINTLEVEL <i>a d</i>). Binding either variable to NIL is equivalent to specifying a value of -1 for the corresponding argument to PRINTLEVEL , i.e., it specifies infinite depth or length. Note that in Interlisp, print level is "triangular"—the print length decreases as the depth increases. In Common Lisp, the two are independent. Thus, although print level for both Interlisp and Common Lisp is controlled by a common pair of variables, the Interlisp and Common Lisp print functions interpret them (specifically *PRINT-LENGTH*) slightly differently. In addition, Interlisp observes print level only when printing to the terminal, whereas Common Lisp observes it on all output. |
| *READTABLE* vs SETREADTABLE | Binding *READTABLE* to a readtable specifies the readtable to use in any input/output function with a readtable argument omitted or specified as NIL . Evaluating *READTABLE* is the same as evaluating (GETREADTABLE). There is no longer a "NIL" or "T" readtable in Interlisp. See the discussion of readtables for more details. |

Although the binding style is to be preferred to the **RESETFORM** expression, one difference should be noted with respect to error checking. In a form such as

```
(RESETFORM (RADIX n)  
             some-printing-code)
```

the value of *n* is checked immediately for validity, and an error is signalled if *n* is not an integer between 2 and 36. However, in

```
(LET ((*PRINT-BASE* n))  
      some-printing-code)
```

there is no error checking at the time of the binding; rather, an error will not be signalled until the code attempts to print a number.

Similarly, the values of ***STANDARD-INPUT*** and ***STANDARD-OUTPUT*** must be actual streams, not the values that print functions coerce to streams, such as **NIL**, **T** or window objects.

Integration of Common Lisp and Interlisp Input/output Functions

Common Lisp and Interlisp have slightly different rules for reading and printing, regarding such things as escape characters, case sensitivity and number format. Each has two kinds of printing function, an escaped version (intended for reading back in) and an unescaped version. In order that Common Lisp and Interlisp programs can more freely intermix, Xerox Lisp isolates most of the reading/printing differences in the readtables used by both languages, rather than in the functions themselves. The exact rules have been chosen as a reasonable compromise between backward compatibility with Interlisp and integration with Common Lisp. This section outlines the details of this integration.

In what follows, the phrase "the readtable" generally refers to the readtable in force for the read or print operation being discussed. Specifically, this means an explicit readtable (other than **NIL** or **T**) passed as readtable argument to an Interlisp function, or else the current binding of ***READTABLE***. See the section on readtables for more details.

Section 25.2 Input Functions

The functions **IL:READ** and **CL:READ**, given the same readtable, interpret an input in exactly the same way. That is, the functions obey Common Lisp syntax rules when given a Common Lisp readtable, and Interlisp syntax when given an Interlisp readtable. Thus, the principal difference between the two is in the functionality provided by **CL:READ**'s extra arguments: end of file handling and the ability to specify that the read is recursive, which is mostly important when reading input containing circular structure references (the **##** and **#=** macros). See *Common Lisp, the Language* for details of **CL:READ**'s optional arguments.

There is one further difference between **IL:READ** and **CL:READ**, in the handling of the terminating character. If the read terminates on a white space character, **CL:READ** consumes the character, while **IL:READ** leaves the character in the buffer, to be read by the next input operation. Thus, **IL:READ** is equivalent to **CL:READ-PRESERVING-WHITESPACE**. This difference is so that Interlisp

code that calls (**READC**) following a (**READ**) of a symbol will behave consistently between Koto and Lyric.

The Interlisp function **SKREAD** now defaults its readtable argument to the current readtable, viz., the value of ***READTABLE***, rather than **FILERDTBL**. This makes it consistent with all the other input functions, and is usually the correct thing, especially with the new reader environments used by the File Manager, but it is an incompatible change from Koto. **SKREAD** is also now implemented using Common Lisp's ***READ-SUPPRESS*** mechanism, which means that, unlike in Koto, it does something reasonable when it encounters read macros.

In the Medley release, reading in bitmaps from files is significantly faster.

Section 25.3 Output Functions

The discussion here is limited to the four basic printing functions: the unescaped and escaped Interlisp printing functions (**IL:PRIN1**, **IL:PRIN2**) and the corresponding Common Lisp functions (**CL:PRINC**, **CL:PRIN1**). All other print functions ultimately reduce to these. For example, **IL:PRINT** calls **IL:PRIN2**; **CL:FORMAT** with the **~S** directive performs a **CL:PRIN1**.

IL:PRIN1 is essentially unchanged from previous releases. It uses no readtable at all, so is unaffected by the value of ***READTABLE***. It can be thought of as implicitly using the INTERLISP readtable.

Roughly speaking, **IL:PRIN2** and **CL:PRIN1** behave the same when given the same readtable. In particular, they both produce output acceptable to either **READ** function given the same readtable. Their minor differences are listed below.

CL:PRINC behaves like **CL:PRIN1**, except that it never prints escape characters or package prefixes. Thus, unlike **IL:PRIN1**, it is affected by the value of ***READTABLE***.

For the benefit of user-defined print functions, **IL:PRIN2** and **CL:PRIN1** bind ***PRINT-ESCAPE*** to **T**, while **IL:PRIN1** and **CL:PRINC** bind it to **NIL**. Thus, the print function can always examine ***PRINT-ESCAPE*** to decide whether it needs to print objects in a way that will read back correctly (Common Lisp user print functions may want to use **CL:WRITE** to pass ***PRINT-ESCAPE*** through transparently; Interlisp functions should choose **IL:PRIN2** or **IL:PRIN1** appropriately).

Printing Differences Between **IL:PRIN2** and **CL:PRIN1**

There are two respects in which the Interlisp print functions (both **IL:PRIN1** and **IL:PRIN2**) differ from the Common Lisp ones, independent of readtable:

Line Length. The Interlisp functions respect the output stream's line length, while the Common Lisp functions all ignore it (they never insert newline characters when output approaches the right margin).

Print Level. The Interlisp functions respect the print level variables only when printing to the terminal (unless **PLVLFILEFLG** is true, see the *Interlisp-D Reference Manual*) or when printing with a Common Lisp readtable, whereas the Common Lisp functions respect them on *all* output.

Internal Printing Functions

Interlisp has several functions (e.g., **NCHARS**, **STRINGWIDTH**, **CHCON**, **MKSTRING**) that operate on the "prin1 pname" of an object, or optionally on its "prin2 pname" when given an extra flag and readtable as arguments. These functions are essentially unchanged in Lyric.

In terms of the discussion above, the "prin1 pname" of an object continues to be the characters that would be produced by a call to **IL:PRIN1** at infinite print level and line length, and with ***PRINT-BASE*** bound to 10 (unless **PRXFLG** is true, see *Interlisp-d Reference Manual*). The "prin2 pname" of an object is the list of characters that would be produced by a call to **IL:PRIN2** (or **CL:PRIN1**) using the specified readtable (or ***READTABLE*** if none is given), again at infinite print level and line length.

Exception: the function **STRINGWIDTH** computes the width of the expression as if it were printed at the current ***PRINT-LEVEL*** and ***PRINT-LENGTH***.

Printing Differences between Koto and Lyric

The Common Lisp and Interlisp printing functions use the same strategy for escaping characters in symbol names. Because of this, symbols may print differently in Lyric than they did in Koto, for two reasons: the use of the Common Lisp multiple escape character, and the escaping of numeric print names. Although the appearance is different, the functionality is the same—symbols are still printed in a way that allows them to be correctly read.

Roughly speaking, the multiple escape character is used to escape symbol names that would require two or more single escape characters. Thus, for example, a symbol that printed as `%(OH% NO%)` in Koto will print in Lyric as `| (OH NO) |`. However, in the old readtables that lack a multiple escape character (e.g., OLD-INTERLISP-T), the single escapes are still used. Multiple escapes are also used to print a symbol containing lower-case letters when the readtable is case-insensitive, e.g., `|Small|` in a Common Lisp readtable. Keep in mind also that some additional characters are now "special", e.g., colon in all new readtables, semi-colon in Common Lisp. Thus, the typical NS File "full name" will be printed with the multiple escape character.

Since it is now possible to create symbols that have "numeric" print names, such symbols must be printed with suitable escape characters, so that on input they are not misinterpreted as numbers. For example, the symbol whose print name is "1.2E3" is printed as `|1.2E3|`. In read tables lacking a multiple escape character, the single escape character is used instead, e.g., `1.2E3` in the old Interlisp T readtable. A print name is considered

numeric according to the definition of "potential number" in Common Lisp (p. 341). Even if such a symbol is not readable in the current system as a number, it still needs to be escaped in case it is read into another system that treats it as numeric (either another Common Lisp implementation, or a future implementation of Xerox Lisp). Thus, some old Interlisp symbols now print escaped where they didn't in Koto; e.g., the **PRINTOUT** directive | .P2 | is a potential number.

Bitmap Syntax

Bitmaps are printed in a new syntax in Lyric. When ***PRINT-ARRAY*** is **NIL** (the default at top level), a bitmap prints in roughly the same compact form as previously:

```
#<BITMAP @ nn,nnnnnn>
```

If ***PRINT-ARRAY*** is **T**, a bitmap prints in a manner that allows it to be read back:

```
#*( Width Height [BitsPerPixel]) XXXXXXXXX...
```

Width and *Height* are measured in pixels; *BitsPerPixel* is supplied for bitmaps of other than the default of 1 bit per pixel. Each *X* represents four bits of a row of the bitmap; the characters @ and A through o are used in this encoding. Thus, there are $4 \lceil \text{Width} * \text{BitsPerPixel} / 16 \rceil$ *X*'s for each row.

MAKEFILE binds ***PRINT-ARRAY*** to **T** so that bitmaps print readably in files. E.g., if the value of **FOO** is a bitmap, the command (VARS FOO) dumps something like

```
(RPAQQ FOO #*(10 10)ADSDKJFDKJH...)
```

Note that with this new format, bitmaps are readable even inside a complex list structure. This means you need no longer use the **UGLYVARS** command when dumping a list containing bitmaps if the bitmaps were previously the only "unprintable" part of the list.

Section 25.8 Readtables

(III:25.34)

The input/output syntaxes of Common Lisp and Interlisp differ in a few significant ways. For example, Common Lisp uses "\" as the escape character, whereas Interlisp uses "%". Common Lisp input is case-insensitive (lower-case letters are read as upper-case), whereas Interlisp is case-sensitive. In Xerox Lisp, these differences are accommodated by having different readtables for the two dialects. Which syntax is used for input or output depends on which readtable is being used (either as an explicit argument to the read/print function or by being the "current" readtable).

Interlisp readtables have been extended to include features of Common Lisp syntax. There is a registry of named readtables to make it easier to choose a readtable. The default Interlisp readtable has been modified to make it look a little closer to Common Lisp.

Also, Lisp has a new mechanism for maintaining read/print consistency. This means that even though Koto files may contain

characters that are now "special", such as colon, you need make no changes to them—the File Manager knows how to load them correctly. See *IRM*, Chapter 17, Reader Environments and File Manager for details of this mechanism.

Differences Between Interlisp and Common Lisp Read Tables

When reading or printing, the readtable dictates the syntax rules being followed. As in past releases, the readtable indicates which characters must be escaped when printing a symbol (and ***PRINT-ESCAPE*** is true). In addition, in Lyric the readtable specifies such things as which escape character to use (\ or %) and the package delimiter to print on package-qualified symbols. The less obvious rules are detailed below.

Printing numbers. Numbers are always printed in the current print base (the value of the variable ***PRINT-BASE***, or equivalently the value of (**RADIX**). Whether to print a radix specifier is determined by the readtable. In Common Lisp, a radix specifier is printed exactly when the value of ***PRINT-RADIX*** is true. The radix specifier is a suffix decimal point in base 10, or a prefix using # for other bases. In Interlisp, a radix specifier is printed if the base is not 10, ***PRINT-ESCAPE*** is true, and the number is not less than the print base. The radix specifier is a suffix Q for octal, or a prefix using # (or | in old Interlisp readtables) for other bases. There is no decimal radix specifier.

Reading numbers. In Common Lisp, numbers are read in the current value of ***READ-BASE***, and a trailing decimal point is interpreted as a decimal radix specifier. In Interlisp, numbers are always read in base 10, and trailing decimal point denotes a floating-point number.

Case conversion. In a case-insensitive readtable (as Common Lisp is), the value of ***PRINT-CASE*** controls how upper-case symbols are printed, and lower-case letters in symbols are escaped. In a case-sensitive readtable (as Interlisp is), ***PRINT-CASE*** is ignored, so all letters in symbols are printed verbatim. ***PRINT-CASE*** is also ignored by **PRIN1**, which implicitly uses an Interlisp readtable.

Ratios. The character slash (/) is interpreted as the ratio marker in all readtables except old Interlisp readtables (specifically, those whose **COMMONNUMSYNTAX** property is **NIL**). This is so that old files containing symbols with slashes are not misinterpreted as ratios. Thus, the characters "1/2" are read in new readtables as the ratio 1/2, but in old Interlisp readtables as the 3-character symbol |1/2| (| is the multiple-escape character, see below). Ratios are printed in old Interlisp readtables in the form |./ numerator denominator|.

Packages. Symbols are interned with respect to the current package (the binding of ***PACKAGE***) except in old Interlisp readtables (specifically, those whose **USESILPACKAGE** property is **T**), where symbols are read with respect to the INTERLISP package, independent of the binding of ***PACKAGE***. Again, this is a backward-compatibility feature: Interlisp had no package system, so programmers were not confronted with the need to read and print in a consistent package environment.

Print Level elision. When ***PRINT-LEVEL*** or ***PRINT-LENGTH*** is exceeded, the printing functions denote elided elements and elided tails by printing "&" and "--" with an Interlisp readable, or "#" and "... " with a Common Lisp readable.

Section 25.8.2 New Readtable Syntax Classes

The following new syntax classes are recognized by **GETSYNTAX** and **SETSYNTAX**:

MULTIPLE-ESCAPE This character inhibits any special interpretation of all characters (except the single escape character) up until the next occurrence of the multiple escape character. In Common Lisp and in the new Interlisp readtables this character is the vertical bar ("|"). For example, |(a)| is read as the 3-character symbol "(a)"; |x|y|z| is read as the 5 character symbol "x|y|z".

There is no multiple escape character in the old Interlisp readtables.

PACKAGEDELIM This character separates a package name from the symbol name in a package-qualified symbol. In Common Lisp and in the new Interlisp readtables this character is colon (":"). In the old Interlisp readtables the package delimiter is control-↑ ("↑↑"); it is not intended to be easily typed, but exists only to have a compatible way to print package-qualified symbols in obsolete readtables. See *Common Lisp, the Language* for details of package specification.

Additional Readtable Properties

Read tables have several additional properties in Xerox Lisp. These are accessible via the function **READTABLEPROP**:

(READTABLEPROP RDTBL PROP NEWVALUE) [Function]

Returns the current value of the property *PROP* of the readtable *RDTBL*. In addition, if *NEWVALUE* is specified, the property's value is set to *NEWVALUE*. The following properties are recognized:

NAME The name of the readtable (a string, case is ignored). The name is used for identification when printing the readtable object itself, and can be given to the function **FIND-READTABLE** to retrieve a particular readtable.

CASEINSENSITIVE If true, then unescaped lower-case letters in symbols are read as upper-case when this readtable is in effect. This property is true by default in Common Lisp readtables and false in Interlisp readtables.

COMMONLISP If true, then input/output obeys certain Common Lisp rules; otherwise it obeys Interlisp rules. This is described in more detail in the section on reading and printing. Setting this property to true also sets **COMMONNUMSYNTAX** true and **USESILPACKAGE** false.

COMMONNUMSYNTAX If true, then the Common Lisp rules for number parsing are followed; otherwise the old Interlisp rules are used. This affects the interpretation of "/" and the floating-point exponent specifiers "d", "f", "l" and "s". It does not affect the interpretation of decimal point and ***READ-BASE***, which are controlled by the **COMMONLISP** property. **COMMONNUMSYNTAX** is true for Common Lisp

readtables and the new Interlisp readtables; it is false for old Interlisp readtables.

USESILPACKAGE

This is a backward compatibility feature. If **USESILPACKAGE** is true, then the Interlisp input/output functions read and print symbols with respect to the Interlisp package, independent of the current value of ***PACKAGE***. This property is true by default for old Interlisp readtables and false for others.

The following properties let the print functions know what characters are being used for certain variable syntax classes so that they can print objects in a way that will read back correctly. Note that it is possible for several characters to have the same syntax on input, but only one of the characters is used for output. Also note that only the three syntax classes **ESCAPE**, **MULTIPLE-ESCAPE** and **PACKAGEDELIM** are parameterized for output; the others (such as **LEFTPAREN** and **STRINGDELIM**) are hardwired—the same character is always used.

ESCAPECHAR

This is the character code for the character to use for single escape. Setting this property also gives the designated character the syntax **ESCAPE** in the readtable.

MULTIPLE-ESCAPECHAR

This is the character code for the character to use for multiple escape. Setting this property also gives the designated character the syntax **MULTIPLE-ESCAPE** in the readtable.

PACKAGECHAR

This is the character code for the package delimiter. Setting this property also gives the designated character the syntax **PACKAGEDELIM** in the readtable.

(FIND-READTABLE NAME)

[Function]

Returns the readtable whose name is *NAME*, which should be a symbol or string (case is ignored); returns **NIL** if no such readtable is registered. Readtables are registered by calling **(READTABLEPROP rdtbl 'NAME name)**.

(COPYREADTABLE RDTBL)

[Function]

COPYREADTABLE has been extended to accept a readtable name as its *RDTBL* argument (the old value **ORIG** could be considered a special case of this). For example, **(COPYREADTABLE "INTERLISP")** returns a copy of the INTERLISP readtable. **COPYREADTABLE** preserves all syntax settings and properties except **NAME**.

Section 25.8 Predefined Readtables

The following readtables are registered in the Lyric release of Lisp:

INTERLISP

This is the "new" Interlisp readtable. It is used by default in the Interlisp Exec and by the File Manager to write new versions of pre-existing source files. It thus replaces the old T readtable, **FILERDTBL**, **CODERDTBL** and **DEDITRDTBL**. It differs from them in the following ways:

| (vertical bar)

has syntax **MULTIPLE-ESCAPE** rather than being used as a variant of the Common Lisp dispatching **#** macro character.

#	is used as the Common Lisp dispatching # macro character. For example, to type a number in hexadecimal, the syntax is #xnnn rather than xnnn.
: (colon)	has syntax PACKAGEDELIM .
' (quote)	reads the next expression as (QUOTE expression).
` (backquote)	are used to read backquoted expressions
, (comma)	

In addition, the Common Lisp syntax for numbers is supported (the readtable has property **COMMUNNUMSYNTAX**). For example, the characters "1/2" denote a ratio, not a symbol. Note, however, that trailing decimal point still means floating point, rather than forcing a decimal read base for an integer.

The syntax for quote, backquote, and comma is the same as in OLD-INTERLISP-T, so you will not see any difference when typing to an Interlisp Exec. However, the fact that files are also written in the new INTERLISP readtable means that the prettyprinter is now able to print quoted and backquoted expressions much more attractively on files (and to the display as well).

LISP This readtable implements Common Lisp read syntax, exactly as described in *Common Lisp, the Language*.

XCL This readtable is the same as LISP, except that the characters with ASCII codes 1 thru 26 have white-space (**SEPRCHAR**) syntax. This readtable is intended for use in File Manager files, so that font information can be encoded on the file.

The following readtables are provided for backward compatibility. They are the same as the corresponding readtables in the Koto release, with the addition of the **USESILPACKAGE** property.

ORIG This is the same as the ORIG readtable described in the *Interlisp-D Reference Manual*. When using a readtable produced by (**COPYREADTABLE** 'ORIG), expressions will read and print exactly the same in Koto and Lyric.

OLD-INTERLISP-FILE This is the same as the FILERDTBL described in the *Interlisp-D Reference Manual*. This readtable is used to read source files produced in the Koto release. Note that in Lyric, FILERDTBL is no longer used when reading or writing new files; see the section on reader environments.

OLD-INTERLISP-T This is the same as the T readtable described in the *Interlisp-D Reference Manual*.

If you wish to change the syntax used by a standard readtable, it is recommended instead that you copy the readtable, give it a distinguished name, and make the change in the new readtable. This will reduce the likelihood that you will try to read another user's files in an incompatible readtable, or that another user will fail reading yours. See chapter 17, Reader Environments and the File Manager, for more details.

Koto Compatibility Considerations

In order to consistently read a data structure that you have previously printed, it is important that **READ** and **PRINT** both use the same readtable and package. Code that calls **READ** or **PRINT** without explicitly specifying a readtable (via the *RD_TBL* argument or by doing a **SETREADTABLE**) is thus in some danger of reading and printing inconsistently.

Specifying Readtables and Packages

In Koto, the "primary" (NIL) readtable was not significantly different from the other Interlisp readtables, and users tended not to make significant modifications to the primary readtable anyway. As a result, it was easy to write code that was not careful about readtables and get away with it. In Lyric, however, there are significant differences among commonly used readtables. Thus, if code using the default readtable called **PRINT** under, say, the Common Lisp Executive and tried to **READ** the expression back while running under an Interlisp Executive, it might very well get inconsistent results.

Lyric also introduces the extra complication of the default package, which is the other important parameter affecting the behavior of **READ** and **PRINT**.

Programmers are thus advised to fix any code that uses **READ** and **PRINT** as a way of storing and retrieving Lisp expressions to be sure to specify a readtable and package environment. For new code in Lyric, this can be done by binding the special variables ***READTABLE*** and ***PACKAGE***. If it is necessary to write code that works in both Koto and Lyric, the programmer should pass an explicit readtable to all **READ** and **PRINT** functions, or set the primary readtable using (**RESETFORM (SETREADTABLE *rdtbl*) --**). If the readtable chosen is either *FILERDTBL* or one derived as a copy of *ORIG*, then **READ** and **PRINT** will automatically use the INTERLISP package in Lyric, thereby avoiding any need to specify a binding for ***PACKAGE***.

The T Readtable

An additional possible incompatibility exists with regard to the Koto T readtable: The T readtable was "the readtable used when reading from the terminal". In Lyric, the T readtable is synonymous with NIL, and all Executives bind ***READTABLE*** to the appropriate value for the Exec. This is unlikely to be a major source of incompatibility, as few programs depend on printing something in the T readtable in a way that needs to read back consistently.

PQUOTE Printed Files

In Lyric, the prettyprinter automatically prints quoted and backquoted expressions attractively. Hence, the PQUOTE Lispusers module is now obsolete. However, if you have written files in the past with the PQUOTE module loaded into your environment, you need to do the following in Lyric in order to load those files:

```
(SETSYNTAX (CHARCODE '"') '(MACRO FIRST READQUOTE)
FILERDTBL)
```

You can then load the old files. New files produced in Lyric by **MAKEFILE** will automatically be loadable, so you need only perform the **SETSYNTAX** change as long as you still have old files written with PQUOTE. Remember, of course, that as long as the **SETSYNTAX** is in effect (as with the old PQUOTE module), if you read old files that were written without PQUOTE you may read them incorrectly.

Back-Quote Facility

The back-quote facility now fully conforms with *Common Lisp the Language*. This means some cases of nested back-quote now work correctly. Back-quote forms are also more attractively displayed by the prettyprinter. Users should beware, however, that the back-quote facility does not attempt to create fresh list structures unless it is necessary to do so. Thus for example,

'(1 2 3)

is equivalent to

'(1 2 3)

not

(LIST 1 2 3)

If you need to avoid sharing structure you should explicitly use **LIST**, or **COPY** the output of the back-quote form.

Chapter 28 Windows and Menus

Section 28.5.1 Menu Fields

(III:28.38)

With the Medley release, multi-column menus can have rollout submenus.

[This page intentionally left blank]

4. CHANGES TO INTERLISP-D IN LYRIC/MEDLEY

NOTE: Chapter 4 is organized to correspond to the original *Interlisp-D Reference Manual*, and explains changes that have occurred in Interlisp-D with the Lyric and Medley releases. To make it easy to use this chapter with the *IRM*, information is organized by *IRM* volume and section numbers. Section headings from the *IRM* are maintained to aid in cross-referencing.

Lyric information as well as Medley release enhancements are included. Medley additions are indicated with revision bars in the right margin.

VOLUME I—LANGUAGE

Chapter 3 Lists

Section 3.2 Building Lists From Left To Right

(I:3.7)

The functions **DOCOLLECT** and **ENDCOLLECT** are no longer supported.

(I:3.8)

The description of the **ADDTOSCRATCHLIST** function has been revised to read:

(ADDTOSCRATCHLIST <i>VALUE</i>)	[Function]
--	------------

For use inside a **SCRATCHLIST** form. *VALUE* is added on to the end of the value being collected by **SCRATCHLIST**. When the **SCRATCHLIST** returns, its value is a list containing all of the things that **ADDTOSCRATCHLIST** has added.

Section 3.10 Sorting Lists

(I:3.17)

(SORT <i>DATE</i> <i>COMPAREFN</i>)	[Function]
--	------------

There is no safe interrupt to **SORT**—if you abort a call to **SORT** by *any* means the possibility exists for losing elements from the list being sorted.

Chapter 6 Hash Arrays

(I:6.1)

(HASHARRAY	MINKEYS OVERFLOW HASHBITSFN EQUIVFN RECLAIMABLE	
REHASH-THRESHOLD)		[Function]

The function **HASHARRAY** has two new optional arguments, *RECLAIMABLE* and *REHASH-THRESHOLD*. If *RECLAIMABLE* is true, then entries in the hash table are considered "reclaimable" in the sense that the system is permitted to remove any key and its associated value from the hash table at any time. In practice, the contract is less severe: the system only removes keys when a hash table fills and is about to be rehashed, and then it only removes keys whose reference count is one, and to which there are thus no pointers outstanding except possibly from the stack (local variables). This is useful for hash tables that serve to cache information about Lisp objects to avoid recomputation; for example, the system hash table **CLISPARRAY** is now reclaimable. Discarding keys keeps the table from necessarily needing to grow, and potentially allows the storage consumed by both the key and value to be reclaimed.

Section 6.1 Hash Overflow

(I:6.3)

You should note changes to the wording of two of the possibilities for the overflow method:

The first sentence for **NIL** should read: The array is automatically enlarged by *at least* a factor of 1.5 every time it overflows.

The explanation for "a positive integer N" should read: The array is enlarged to include *at least* N more slots than it currently has.

Chapter 7 Integer Arithmetic

(I:7.5)

The variables **MIN.INTEGER** and **MAX.INTEGER** have been removed from the *Interlisp-D Reference Manual*. Therefore, calling **(MIN)** and **(MAX)** is an error.

(I:7.7)

(FIXR N)		[Function]
----------	--	------------

When N is exactly half way between two integers, **FIXR** rounds it to the even number. For example **(FIXR 1.5)** \Rightarrow 2 and **(FIXR 2.5)** \Rightarrow 2.

Section 7.3 Logical Arithmetic Functions

The function **INTEGERLENGTH** does *not* coerce floating point numbers to integers; rather, it signals an error, "Arg not Integer". (This was true in Koto as well.)

Section 7.5 Other Arithmetic Functions

(I:7.13)

The algorithms for **SIN**, **COS**, and other trigometric functions have been tuned and are now accurate to at least six significant figures.

Chapter 8 Record Package

(I:8.11)

When using **BLOCKRECORD**, it is an error to try to declare a record with a zero-length field. Previously, the system would go into an infinite loop. In the Medley release, the system will now detect this and signal an error.

Chapter 9 Conditionals and Iterative Statements

Section 9.2 Equality Predicates

(I:9.3)

(EQUALALL X Y)

[Function]

Add the following NOTE to the **EQUALALL** function:

Note: In general, **EQUALALL** descends all the way into all datatypes, both those defined by the user and those built into the system. If you have a data structure with fonts and pointers to windows, **EQUALALL** will descend into those also. If the data structures are circular, as windows are, **EQUALALL** can cause a Stack Overflow error.

Section 9.8.3 Condition I.s. oprs

UNTIL N (N a number)

[I.S. Operator]

REPEATUNTIL N (N a number)

[I.S. Operator]

These descriptions were included in the *Interlisp-D Reference Manual* in error and should be removed. **UNTIL** and **REPEATUNTIL** work *only* with predicate expressions, not numbers.

Chapter 10 Function Definition, Manipulation , and Evaluation

Section 10.2 Defining Functions

(I:10.11)

In the definition of the **MOVD** function, the sentence "**COPYDEF** is a higher-level function that only moves expr definitions, but..." should be revised to read:

COPYDEF is a higher-level function that not only moves expr definitions, but also works for variables, records, etc.

Section 10.5 Functional Arguments

(I:10.19)

FUNARG functionality (non-**NIL** second argument to **FUNCTION**) has been withdrawn. Most of the uses for Interlisp **FUNARG**'s are better written using the lexical closure functionality of Common Lisp.

Section 10.6.2 Interpreting Macros

The variables **SHOULDCOMPILEMACROATOMS** and **UNSAFEMACROATOMS** no longer exist.

Chapter 11 Variable Bindings and the Interlisp Stack

(II:11.2)

In Lisp there is a fixed amount of space allocated for the stack. When this space is exhausted, the **STACK OVERFLOW** error occurs. However, if the system waited until the stack were *really* exhausted, there wouldn't be room to run the debugger. Thus, a portion of the stack space is reserved; when the stack intrudes into the reserved area, it causes a stack overflow interrupt, and subsequently a call to the debugger.

In order not to get a **STACK OVERFLOW** error while inside the debugger, this intrusion into the reserved area is only noted once. If the reserved area is exhausted, then a "hard" stack overflow occurs (a 9319 MP halt), from which the only recourse is a hard reset via **STOP** (or Ctrl-D from TeleRaid). Following a hard reset, the stack is cleared, stack overflow detection is reenabled, and all processes are restarted.

The implications of this are that you should not attempt any deep computations while inside the debugger for a stack overflow error, and you should call **(HARDRESET)** as soon as possible in order that subsequent stack overflows can again be caught in the debugger before they advance to the MP halt. In order to make this more convenient, the system automatically calls **(HARDRESET)** if you exit the debugger via the **^** or **OK** commands, or abort with a Ctrl-D. The only way to exit the debugger without having a

(HARDRESET) occur is by using the **RETURN** command. You can disable this feature by setting **AUTOHARDRESETFLG** to **NIL**, in which case you must be sure to perform the **(HARDRESET)** yourself if you want the next stack overflow to be detected early enough to enter the debugger.

Section 11.2.1 Searching the Stack

(STKPOS FRAMENAME N POS OLDPOS)	[Function]
--	------------

(STKPOS 'STKPOS) does not cause an error; it merely returns NIL. (This was true in Koto as well.) It is still not permissible to create a pointer to the active frame; however, **STKPOS** never attempts to, as it starts searching for the specified frame by looking first at its caller.

Section 11.2.2 Variable Bindings in Stack Frames

(I:11.7)

(STKARG N POS —)	[Function]
-------------------------	------------

(STKNARGS POS —)	[Function]
-------------------------	------------

The functions **STKARG** and **STKNARGS** will now return the number of arguments supplied to a Lambda Nospread when there is a break. The **?=** command will show all the arguments.

(SETSTKARGNAME N POS NAME)	[Function]
-----------------------------------	------------

The function **SETSTKARGNAME** does not work for interpreted frames.

Section 11.2.5 Releasing and Reusing Stack Pointers

(CLEARSTK FLG)	[Function]
-----------------------	------------

(CLEARSTK NIL) is a no-op—the ability to clear all stack pointers is inconsistent with the modularity implicit in a multi-processing environment.

CLEARSTKLST	[Variable]
--------------------	------------

NOCLEARSTKLST	[Variable]
----------------------	------------

The variables **CLEARSTKLST** and **NOCLEARSTKLST** are no longer used. (More precisely, they are used only by the Old Interlisp Executive, which means that programs can no longer depend on them.)

Section 11.2.7 Other Stack Functions

(II:11.13)

In the **REALFRAMEP** function, the **INTERPFLG** argument description has been corrected to read:

If **INTERPFLG=T** returns **T** if **POS** is not a dummy frame. For example, if **(STKNAME POS)=COND**, **(REALFRAMEP POS)** is **NIL**, but **(REALFRAMEP POST)** is **T**.

Chapter 12 Miscellaneous

Section 12.2 Idle Mode

The following properties in **IDLE.PROFILE** are new or have meanings different from the documentation in the *Interlisp-D Reference Manual*:

ALLOWED.LOGINS

The authentication aspects of this property have been separated into the **AUTHENTICATE** property. The value of this property now speaks specifically to who is allowed to exit idle mode: If the value is **NIL** (or any other non-list), no login at all is required to exit Idle mode. Otherwise, the value is a list composed of any of the following:

***** Require login, but let anyone exit idle mode. This will overwrite the previous user's name and password each time idle mode is exited.

T Let the previous user (as determined by **USERNAME**) exit idle mode. If the user name has not been set, this is equivalent to *****.

A user name Let this specific user exit idle mode.

A group name Allow any members of this group (an NS Clearinghouse group name) to exit idle mode.

The initial value for **ALLOWED.LOGINS** is **(T *)**, i.e., anyone is allowed to exit idle mode.

AUTHENTICATE

The value of this property determines what mechanism is used to check passwords. If **T**, use the NS authentication protocol (requires the presence of an NS Authentication server accessible via the network). If **NIL**, do not check the password at all—accept any password. This is only particularly useful if **ALLOWED.LOGINS** contains *****.

The initial value of **AUTHENTICATE** is **T**.

FORGET

If this is the symbol **FIRST**, the user's password will be erased when idle mode is entered. Otherwise, this property is relevant only when **ALLOWED.LOGINS** is **NIL** (if **ALLOWED.LOGINS** is a list, then some sort of login is required, which will have the effect of erasing any previous login): If **FORGET** is non-**NIL**, the user's password will be erased when idle mode is exited. Initial value is **T** (erase password on exit).

Note: If the password is erased on *entry* to Idle mode (value **FIRST**), any program left running when idle mode is entered may fail if it tries doing anything requiring passwords (such as accessing file servers).

LOGIN.TIMEOUT

Value is a number indicating how many seconds Idle's prompt for a login should remain up before timing it out and resuming Idle mode. Initial value is 30. This feature avoids the problem of having an Idle machine "freeze up" indefinitely (stop running the idle pattern) just because someone brushed against the keyboard.

RESETVARS

This property is no longer used; rather, the value of the global variable **IDLE.RESETVARS** is used instead.

SUSPEND.PROCESS.NAMES

This property is no longer used; rather, the value of the global variable **IDLE.SUSPEND.PROCESS.NAMES** is used instead.

Section 12.3 Saving Virtual Memory State

AROUNDEXITFNS

[Variable]

This variable provides a way to "advise" the system on cleanup and restoration activities to perform around **LOGOUT**, **SYSOUT**, **MAKESYS** and **SAVEVM**; it subsumes the functionality of **BEFORESYSOUTFORMS**, **AFTERLOGOUTFORMS**, etc. Its value is a list of functions (names) to call around every "exit" of the system. Each function is called with one argument, a symbol indicating which particular event is occurring:

BEFORELOGOUT

The system is about to perform a **LOGOUT**. The event function might want to save state, notify a network connection that it is about to go away, etc.

BEFORESYSOUT
BEFOREMAKESYS
BEFORESAVEVM

The system is about to perform a **SYSOUT**, **MAKESYS**, or **SAVEVM**. Often these three events are treated equivalently; however, sometimes the distinction is interesting. For example, a program might want to perform a much more extensive tidying-up before **MAKESYS** than if it is merely doing a routine **SAVEVM**.

AFTERLOGOUT
AFTERSYSOUT
AFTERMAKESYS
AFTERSAVEVM

The system is starting up a virtual memory image that was saved by performing a **LOGOUT**, **SYSOUT**, etc. Ordinarily, the event function should treat all of these the same—in all four cases, some arbitrary amount of time has passed, remote files may have come and gone, a different user may be logged in, or the virtual memory image might even be running on a different workstation.

AFTERDOSYSOUT
AFTERDOMAKESYS
AFTERDOSAVEVM

The system is continuing in the same virtual memory image following a **SYSOUT**, **MAKESYS**, or **SAVEVM** (as opposed to having just booted the same virtual memory image). Ordinarily, these events are uninteresting; they exist solely so that actions taken by the **BEFORExxx** events can be compensated for after the event. For example, if the before event cleared a cache, the after event might initiate refilling it. There is, of course, no event **AFTERDOLOGOUT**, as **LOGOUT** does not "continue".

Section 12.4 System Version Information

(I:12.13)

In the description of the **MACHINETYPE** function, add another machine, the **DOVE** (for the Xerox 1186).

VOLUME II—ENVIRONMENT

Chapter 13 Interlisp Executive

(I:23.37)

(**READLINE** *RD_TBL* — —)

[Function]

The *Interlisp-D Reference Manual* states:

The description on p 13.37 of **READLINE**'s behavior when one or more spaces precede the carriage return applies only when **LISPXREADFN** is **READ**. **LISPXREADFN** is initially **TTYINREAD**, which ignores spaces before the carriage return, and thus will never prompt you with "..." for an additional line. Also, the new Executive does not use **READLINE** at all, so you will never see this behavior in a new Executive, independent of the setting of **LISPXREADFN**.

Chapter 14 Errors and Breaks

Section 14.5 Break Window Variables

(II:14.15)

Setting the variable **BREAKREGIONSPEC** to **NIL** no longer creates problems if there is a subsequent break.

Section 14.8 Catching Errors

(II:14.22)

The **Nlambda** functions **ERSETQ** and **NLSETQ** now allow evaluation of an arbitrary number of forms, rather than only one.

(II:14.26)

For Medley, the Interlisp interpreter's handler for **RESETFORM** has been fixed (in Lyric, it worked only from the Common Lisp interpreter, or compiled) .

Chapter 17 File Package

Note: The File Package is now known as the File Manager.

Section 17.8.1 Functions for Manipulating Typed Definitions

(II:17.26)

(HASDEF NAME TYPE SOURCE SPELLFLG) [Function]

Clarification: **HASDEF** for type FNS (or NIL) indicates that *NAME* has an editable source definition, not that *NAME* is defined at all. Thus if *NAME* exists on a file for which you have loaded only the compiled version but not the source, **HASDEF** returns **NIL**.

Section 17.8.2 Defining New File Package Types

(II:17.31)

In the **WHENCHANGED** File Package Type Property the *REASON* argument passed to **WHENCHANGED** no longer is **T** or **NIL**. The Note has been revised as follows:

Note: The *REASON* argument passed to **WHENCHANGED** functions is either **CHANGED** or **DEFINED**.

(II:17.32)

The Nospread Function **FILEPKGTYPE** returns a property list rather than an alist.

Section 17.9.2 Variables

(II:17.36)

In the Lyric release, **HORRIBLEVARS** did not preserve common substructures across several variables.

In Lyric, you could not dump an **UGLYVARS** or **HORRIBLEVARS** whose printed representation required more than *ARRAY-TOTAL-SIZE-LIMIT* characters. This is no longer the case with the Medley release.

Section 17.9.8 Defining New File Package Commands

(II:17.47)

The Nospread Function **FILEPKGCOM** returns a property list rather than an alist.

Section 17.11 Symbolic File Format

(PRETTYDEF PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES) [Function]

PrettyDEF accepts only a symbol for its file argument.

In Lyric and Medley, **SYSPRETTYFLG** is ignored in Interlisp exec and does not pretty-print values in the executive.

(LISPSOURCEFILE FILE)[Function]

LISPSOURCEFILE is more specifically defined to mean that the file is in File Manager format *and* has a file map.

Section 17.11.3 File Maps

File maps are no longer stored on the FILEMAP property. See **GET-ENVIRONMENT-AND-FILEMAP** in the section entitled "Programmer's Interface to Reader Environments" in chapter 3.

Chapter 18 Compiler

CAUTION: Files compiled in Medley cannot be loaded back into Lyric. Medley-compiled .LCOM and .DFASL files will produce an error message when loaded into Lyric. (Lyric-compiled .LCOM and .DFASL files can be loaded and run in Medley.) If you need to run a Medley file in Lyric, load the source file and use the Lyric compiler.

Note that you should not attempt to compile a file containing a function named **STOP**. The format of the .LCOM file produced by **BCOMPL** or **TCOMPL** admits an unfortunate ambiguity in the treatment of the symbol **STOP—LOAD** prefers to treat it as the token signifying the end of the file, rather than as starting the definition of the function **STOP**.

There is no such restriction for the .DFASL files produced by **CL:COMPILE-FILE**.

Chapter 21 CLISP

Section 21.8 Miscellaneous Functions and Variables

(CLEARCLISPARRAY NAME — —)[Function]

Macro and CLISP expansions are cached in CLISPARRAY, the system's CLISP hash array. When anything changes that would invalidate an expansion, it needs to be removed from the cache. CLEARCLISPARRAY removes from the CLISP hash array any key whose CAR is NAME. The system does this automatically whenever you edit a clisp or macro form, or when you redefine a clisp word or macro definition. However, you may need to call CLEARCLISPARRAY explicitly if you change something in a more subtle way, e.g., you redefine a function used by a macro. If your change invalidates an unknown set of expansions, you might prefer to take the performance penalty of calling (CLRHASH CLISPARRAY) to invalidate the entire cache, just to make sure no incorrect expansions are kept around.

Chapter 22 Performance Issues

Section 22.1 Storage Allocation and Garbage Collection

The following should be appended to the description of garbage collection in Interlisp-D:

Another limitation of the reference-counting garbage collector is that the table in which reference counts are maintained is of fixed size. For typical Lisp objects that are pointed to from exactly one place (e.g., the individual conses in a list), no burden is placed on this table, since objects whose reference count is 1 are not explicitly represented in the table. However, large, "rich" data structures, with many interconnections, backward links, cross references, etc, can contribute many entries to the reference count table. For example, if you created a data structure that functioned as a doubly-linked list, such a structure would contribute an entry (reference count 2) for each element.

When the reference count table fills up, the garbage collector can no longer maintain consistent reference counts, so it stops doing so altogether. At this point, a window appears on the screen with the following message, and the debugger is entered:

```
Internal garbage collector tables have overflowed, due
to too many pointers with reference count greater than 1.
*** The garbage collector is now disabled. ***
Save your work and reload as soon as possible.
```

[This message is slightly misleading, in that it should say "count not equal to 1". In the current implementation, the garbage collection of a large pointer array whose elements are not otherwise pointed to can place a special burden on the table, as each element's reference count simultaneously drops to zero and is thus added to the reference count table for the short period before the element is itself reclaimed.]

If you exit the debugger window (e.g., with the RETURN command), your computation can proceed; however, the garbage collector is no longer operating. Thus, your virtual memory will become cluttered with objects no longer accessible, and if you continue for long enough in the same virtual memory image you will eventually fill up the virtual memory backing store and grind to a halt.

Section 22.5 Using Data Types Instead of Records

(II:22.13)

The note in this section states that "pages for datatypes are allocated one page at a time." The note should read:

Space for datatypes is allocated two pages at a time. Thus, each datatype for which any instances at all have been allocated has at least two pages assigned to it.

Chapter 23 Processes

Section 23.1 Creating and Destroying Processes

(III:23.2)

ADD.PROCESS no longer coerces the process name to a symbol. Rather, process names are treated as case-insensitive strings. Thus, you can use strings for process names, and when typing process commands to an exec, you need not worry about getting the alphabetic case correct.

Section 23.2 Process Control Constructs

The Medley release fixes the **PROCESS.EVAL** and **PROCESS.APPLY** functions. In **PROCESS.EVAL** and **PROCESS.APPLY**, with argument **WAITFORRESULT = T**, if the computation in the other process aborts (or the process is killed), then **PROCESS.EVAL** and **PROCESS.APPLY** return **:ABORTED** instead of hanging.

Section 23.6 Typein and the TTY Process

BACKGROUNDFN

[Variable]

A list of functions to call "in the background". The system runs a process (called "BACKGROUND") whose sole task is to call each of the functions on the list **BACKGROUNDFN** repeatedly. Each element is the name of a function of no arguments. This is a good place to put cheap background tasks that only do something once in a while and hence do not want to spend their own separate process on it. However, note that it is considered good citizenship for a background function with a time-consuming task to spawn a separate process to do it, so that the other background functions are not delayed.

TTYBACKGROUNDFN

[Variable]

This list is like **BACKGROUNDFN**, but the functions are only called while in a tty input wait. That is, they always run in the tty process, and only when the user is not actively typing. For example, the flashing caret is implemented by a function on this list. Again, functions on this list should spend very little time (much less than a second), or else spawn a separate process.

Section 23.8 Process Status Window

The Medley release modifies the way in which the Process Status Window can be reshaped and refreshed.

The Process Status Window is now created in such a way that reshaping the window reshapes **ONLY** the backtrace window, not the main window.

The process status window now refreshes itself automatically following a KILL command.

VOLUME III—INPUT/OUTPUT

Chapter 24 Streams and Files

Section 24.7 File Attributes

(**GETFILEINFO** *FILE ATTRIB*)

[Function]

NS file servers implement the following additional attributes for **GETFILEINFO** (neither of these attributes is currently settable with **SETFILEINFO**):

READER	The name of the user who last read the file.
PROTECTION	A list specifying the access rights to the file. Each element of the list is of the form (<i>name nametype . rights</i>), where <i>name</i> is the name of a user or group or a name pattern, and <i>rights</i> is one or more of the symbols ALL READ WRITE DELETE CREATE MODIFY. For servers running Services release 10.0 or later, <i>nametype</i> is the symbol "--"; in earlier releases it is either INDIVIDUAL or GROUP, to distinguish the type of name. For example, the value ((Jane Jones: -- ALL) (*: -- READ)) means that user Jane Jones has full access to the file, while all members of the default domain only have read access to the file.

Section 24.9 Local Hard Disk Device

(III:24.22)

In the Medley release, the {DSK} device now accepts a wider range of characters in file names. Almost any character in char set 0 is acceptable. Previously, if you tried to create a file whose name included, for example, an underscore, you would see a "FILE NOT FOUND" error.

Section 24.10 Floppy Disk Device

(III:24.26)

As of the Lyric release, CPM-format floppy disks are no longer supported.

Section 24.12 Temporary Files and CORE Device

(III:24.30)

In Medley, (**GETFILEINFO** xx 'LENGTH) works for both opened and closed **NODIRCORE** streams.

A closed **NODIRCORE** stream can be reopened.

Section 24.18.1 Pup File Server Protocols

UNIXFTPFLG

[Variable]

When the Leaf protocol was first implemented for the Vax Unix operating system, its use was inconsistent with the operation of the Pup FTP server on the same host: the Leaf server supported versions, but the Ftp server knew only about the native, versionless file system. Thus, Lisp could not use the two protocols interchangeably. For example, if it used Ftp to write a file FOO, the Ftp server would, in versionless style, overwrite the versionless file FOO, rather than create a new version FOO;6 to supersede the highest version FOO;5 created by the Leaf server.

Lisp thus makes the conservative assumption that the Ftp server is unusable for anything other than directory enumeration on a host of type UNIX. This is unfortunate, since it is often the case that Ftp is more efficiently implemented than Leaf, since one need only tune the performance of sequential access.

More recent versions of the Unix Pup software have a Leaf and Ftp server more in agreement with each other. Setting **UNIXFTPFLG** to true (it is initially NIL) informs Lisp that all the Unix servers accessible on your internetwork that possess Ftp servers are safe to use in parallel with their Leaf servers.

Section 24.18.1 and 24.18.2 Use of BREAKCONNECTION with File Servers

(III:24.37)

In Medley, the function **BREAKCONNECTION** can be used equally well with NS servers and Leaf servers. Formerly, it only worked on Leaf servers, and there was a separate function (BREAK.NSFILING.CONNECTION *HOST*) to handle NS servers.

(BREAKCONNECTION *HOST FAST*)

[Function]

Breaks the file server connection to *HOST*. If *HOST* = T, breaks connections to all file servers that understand the **BREAKCONNECTION** method (currently Leaf and NS). **BREAKCONNECTION** returns the server name, or if *HOST* = T, returns a list of all hosts that responded to the **BREAKCONNECTION** request.

This function may be useful if Lisp and the server disagree about what files are open, or if the Lisp system is caching something that you do not want it to; e.g., if you get a file busy error from another workstation for a file that you may have touched on this workstation.

The behavior of **BREAKCONNECTION** is server-specific. On an NS server, **BREAKCONNECTION** releases any locks that Lisp may have on recently-accessed files, including those for open files, but does not close any files from Lisp's point of view--any subsequent access to an open file will quietly reestablish the connection. Most NS servers have a short timeout on the order of 10 minutes after which an implicit **BREAKCONNECTION** occurs if you have no files open.

On a Leaf server, **BREAKCONNECTION** first closes any files. If the argument *FAST* is true, it marks the files closed without attempting to close them cleanly. Leaf connections ordinarily do not timeout if any files at all are open.

Section 24.18.2 NS File Server Protocols

(III:24.37)

Medley incorporates the random access capability on NS servers provided by the NSRANDOM LispUsers module in Lyric.

The Medley release also supports NS file names containing characters other than character set 0 (e.g., Greek characters).

Section 24.18.3 Operating System Designations

DEFAULT.OSTYPE

[Variable]

If a host's name is not found in **NETWORKOSTYPES**, its operating system type is assumed to be the value of **DEFAULT.OSTYPE**. This variable may be of use to sites with many servers all of the same type. Its default value (IFS) is, unfortunately, inappropriate for most sites. It is recommended you set **DEFAULT.OSTYPE** in the initialization file that lives on the local disk (*not* in an init file on a file server, since Lisp needs to know the operating system type before talking to the server).

Chapter 25 Input/Output Functions

Section 25.2 Input Functions

(LASTC FILE)

[Function]

The function **LASTC** can return an incorrect result when called immediately following a **PEEKC** on a file that contains run-coded NS characters.

Section 25.3.2 Printing Numbers

(III:25.15)

In the **PRINTNUM** function, the **FLOAT** format option (**FLOAT 7 2 NIL T**) is illegal; change the option to (**FLOAT 7 2 NIL 0**).

Section 25.3.4 Printing Unusual Data Structures

(HPRINT EXPR FILE UNCIRCULAR DATATYPESEEN)

[Function]

Using **HPRINT** to save structures that include pointers to raw storage will cause stack overflows. This includes dumping things using the **VARS**, **UGLYVARS**, or **HORRIBLEVARS** filemanager commands.

For example, a font descriptor points to raw storage, and cannot be dumped; for that reason, other system data types (e.g. windows) that point to fonts also cannot be dumped.

Section 25.4 Random Access File Operations

(III:25.20)

The first argument in the **FILEPOS** function should be called *STR* not *PATTERN*.

(III:25.20)

In the Medley release, the function **COPYBYTES** now accepts *START* and *END* arguments even when the input stream is not random access. This caused an error in earlier releases.

Section 25.6 PRINTOUT

(III:25.27)

The PRINTOUT command **.FONT** changes the **DSPFONT** font permanently, that is, even after printout finishes.

Section 25.8.3 READ Macros

(III:25.42-43)

These **READMACROS** appear only in the OLD-INTERLISP-T readtable. (See Section 2 for a description of Lyric readtables.)

Chapter 26 User Input/Output Packages

Section 26.3 ASKUSER

(ASKUSER WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG
OPTIONSLST FILE)

[Function]

ASKUSER does not accept a string to mean a stream open on the string; you must call **OPENSTRINGSTREAM** if that's what you mean.

Section 26.4 TTYIN Display Typein Editor

(III:26.22)

The following fixes have been made to TTYIN in the Medley release:

- TTYIN now respects the **DSPLEFTMARGIN** of the ttydisplaystream, rather than assuming it is zero.
- You can now assign the keyaction 194 (octal 302--acute accent in the NS character set) to a key and TTYIN will not treat it like the UNDO key (except on the 1132, where this functionality is still on blank-middle).

- TTYIN correctly handles prompts that are wider than the window.
- TTYIN now handles NS characters correctly when you are using a fixed-width font into which you have coerced, say, Classic characters for the non-zero character sets.
- TTYIN now handles Escape completion much more efficiently. If the completion is ambiguous, it completes the unambiguous prefix (as it did in Koto but not Lyric); it also correctly interprets escape characters. For example, in an exec with Common Lisp readtable, it correctly completes symbols that start with \, or a mixed-case symbol written with vertical bars. Also, Escape completion computes character widths correctly when it lowercases an upper case string, rather than leave some garbage bits on the display.
- The off-by-the-descent bug wherein TTYIN sometimes left stray bits at the bottom of the window has been fixed.

Section 26.4.3 Display Editing Commands

(III:26.25)

?= and Meta-P no longer hang if you had an unbalanced string quote in the input.

?, Meta-P, and the **FIX** command now work correctly when there are NS characters in the input.

The printout for ?= is now improved; it respects *print-case*, matches up keywords better, and prints abstract syntax descriptions (such as for cl:do) a bit more clearly.

SMARTARGLIST fetches the argument lists of cl:compiled functions, so ?= now works in more cases.

The Ctrl-X command, when the caret is already positioned at the end of the input and everything but parentheses are balanced (i.e., no unbalanced string quotes or vertical bars), types as many closing parentheses as necessary to complete the input and then returns, much as if you had typed right bracket ("]") in Interlisp. Thus, if the cursor is somewhere in the middle of the input, typing two Ctrl-X's is sufficient to complete (assuming all you needed to type were some more parens).

TTYIN can now be used as a substitute for PROMPTFORWARD. The new function TTYINPROMPTFORWARD takes the same set of arguments as PROMPTFORWARD. In the most common cases it then calls TTYIN in "promptforward" mode, so that you can use the mouse and other TTYIN commands on the input. For cases it can't handle, it calls the old PROMPTFORWARD. These cases are: DONTCHOTYPEIN.FLG or KEYBD.CHANNEL is non-NIL; ECHO.CHANNEL is not a displaystream; or TERMINCHARS.LST contains a character other than cr, space or ^X and you have set the variable TTYIN.USE.EXACT.CHARS (initially NIL) to T. TTYIN saves the old definition of PROMPTFORWARD, so you can either have your program explicitly call TTYINPROMPTFORWARD instead of PROMPTFORWARD, or you can have all calls to

PROMPTFORWORD changed by doing a (MOVD 'TTYINPROMPTFORWORD 'PROMPTFORWORD).

Section 26.4.5 Useful Macros

(III:26.29)

CTRLUFLG is no longer supported by default. To use this feature, turn it on explicitly: (**INTERRUPTCHAR (CHARCODE ^U) 'CTRLUFLG**).

Chapter 27 Graphic Output Operations

Section 27.1.3 Bitmaps

Note: The printed representation of bitmaps has changed. Please see release notes Chapter 3, Integration of Interlisp-D/ Common Lisp, "Bitmap Syntax."

(III:27.4)

The following function has been added to Bitmap Operations between the functions **EXPANDBITMAP** and **SHRINKBITMAP**:

(ROTATE-BITMAP BITMAP) [Function]

Given an m-high by n-wide bitmap, this function returns an n-high by m-wide bitmap. The returned bitmap is the image of the original bitmap, rotated 90 degrees clockwise.

(III:27.4)

In the Medley release, the **EDITBM** function is substantially faster with the inclusion of **FASTEDITBM** (a former LispUsers module) in the sysout.

Section 27.3 Accessing Image Stream Fields

The following functions were not documented in the Koto release of the *Interlisp-D Reference Manual*:

(DSPCLEOL XPOS YPOS HEIGHT) [Function]

"Clear to end of line". Clears a region from (XPOS,YPOS) to the right margin of the display, with a height of HEIGHT. If XPOS and YPOS are **NIL**, clears the remainder of the current display line, using the height of the current font.

(DSPRUBOUTCHAR DS CHAR X Y TTBL) [Function]

Backs up over character code CHAR in the display stream DS, erasing it. If X, Y are supplied, the rubbing out starts from the position specified. **DSPRUBOUTCHAR** assumes CHAR was printed with the terminal table TTBL, so it knows to handle control characters, etc. TTBL defaults to the primary terminal table.

Section 27.6 Drawing Lines

(III:27.17)

The non-**NIL** value of the *DASHING* argument of **DRAWLINE** uses **LINEWITHBRUSH**. **LINEWITHBRUSH** is a width-by-width brush which draws then lifts.

In the Medley release, when using the color argument, Interpress **DRAWLINE** treats 16x16 bitmaps or negative numbers as shades/textures. Positive numbers continue to refer to color maps, and so cannot be used as textures. To convert an integer shade into a negative number use **NEGSHADE** (e.g. (**NEGSHADE** 42495) is -23041).

(III:27.18)

The **RELDRAWTO** function has been corrected so that it no longer draws a spot if the *DX* and *DY* arguments are 0.

Section 27.7 Drawing Curves

(III:27.18)

For the brush width value of **NIL**, the previous default value (**ROUND 1**) has been changed. The default value for the brush width value **NIL** is the **DSPSCALE** of the stream (that is, 1 printer's point wide).

(III:27.19)

A new image stream function, **DRAWARC**, follows **DRAWCIRCLE** in the *InterLisp-D Reference Manual*.

(DRAWARC <i>CENTERX</i> <i>CENTERY</i> <i>RADIUS</i> <i>STARTANGLE</i> <i>NDEGREES</i> <i>BRUSH</i> <i>DASHINGSTREAM</i>)	[Function]
--	------------

Draws an arc of the circle whose center point is (*CENTERX* *CENTERY*) and whose radius is *RADIUS* from the position at *STARTANGLE* degrees for *NDEGREES* number of degrees. If *STARTANGLE* is 0, the starting point will be (*CENTERX* (*CENTERY* + *RADIUS*)). If *NDEGREES* is positive, the arc will be counterclockwise. If *NDEGREES* is negative, the arc will be clockwise. The other arguments are interpreted as described in **DRAWCIRCLE**.

Section 27.8 Miscellaneous Drawing and Printing Operations

(III:27.20)

To have a filled polygon print correctly, set the global variable **PRINTSERVICE** to floating point value 9.0 for printers running Services 9.0 or later.

When using **FILLPOLYGON** to be sent to Xerox 8044 Interpress printers, the global variable **PRINTSERVICE** must be set to the same value as the Print Service installed on your printer, currently either 8.0, 9.0 or 10.0. Thus, if your printer is running Print Service 9.0, you must set the global variable **PRINTSERVICE** to the floating

point value 9.0. This works around an incompatible change in the Xerox 8044 Interpress implementation.

In Medley, Interpress curves are now rendered at a lower accuracy, allowing faster hardcopy. The spline is now rendered at 1/150 inch; in Lyric it was 1/300 inch.

The following function was omitted from previous version of the *Interlisp-D Reference Manual*:

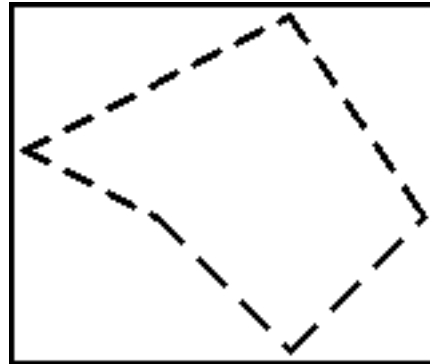
(DRAWPOLYGON POINTS CLOSED BRUSH DASHING STREAM) [Function]

Draws a polygon on the image stream *STREAM*. *POINTS* is a list of positions to which the figure will be fitted (the vertices of the polygon). If *CLOSED* is non-NIL, then the starting position is specified only once in *POINTS*. If *CLOSED* is NIL, then the starting vertex must be specified twice in *POINTS*. *BRUSH* and *DASHING* are interpreted as described in Chapter 27 of the *Interlisp-D Reference Manual*.

For example,

```
(DRAWPOLYGON '((100 . 100) (50 . 125)
               (150 . 175) (200 . 100) (150 . 50))
              T '(ROUND 3) '(4 2) XX)
```

would draw a polygon like the following on the display stream XX.



(III:27.20)

The function **FILLPOLYGON** contains two new arguments, *OPERATION* and *WINDNUMBER*. The new form for the function, and definitions for added arguments, follow.

(FILLPOLYGON POINTS TEXTURE OPERATION WINDNUMBER STREAM) [Function]

OPERATION is the **BITBLT** operation (see page 27.15 in the *Interlisp-D Reference Manual*) used to fill the polygon. If the *OPERATION* is **NIL**, the *OPERATION* defaults to the *STREAM* default *OPERATION*.

WINDNUMBER is the number for the winding rule convention. This number is either 0 or 1; 0 indicates the "zero" winding rule, 1 indicates the "odd" winding rule.

When filling a polygon, there is more than one way of dealing with the situation where two polygon sides intersect, or one polygon is fully inside the other. Currently, **FILLPOLYGON** to a display stream uses the "odd" winding rule, which means that intersecting polygon sides define areas that are filled or not filled somewhat like a checkerboard. For example,

```
(FILLPOLYGON '((125 . 125) (150 . 200) (175 . 125)
              (125 . 175) (175 . 175))
              GRAYSHADE WINDOW)
```

would produce a display something like this:



This fill convention also takes into account all polygons in *POINTS*, if it specifies multiple polygons.

Section 27.12 Fonts

A revised set of font printing metrics is a part of the Lyric release of Lisp. Note that Koto font files are still available to users who request them.

With the revised font set the interline spacing (line leading) is now consistent across all fonts within a point size. Previously, text with multiple fonts (but with the same point size, i.e., if a word were made bold or italic, or if the family were changed) would have different leading on different lines. The new .WD files clean up document appearance.

Note that these printer metric changes affect only hardcopy, not the display. The contents of the display fonts are essentially unchanged in Lyric.

Generally, line leading in the Lyric font files is tighter than in previous releases of the fonts. The default line leading is now the same as the font's nominal point size. As a consequence of the above, any text file (one not already formatted for Interpress) which is printed after installation of the new fonts will be formatted to a different length. This means that decisions regarding TEdit line leading, widows and orphans, left/right pages, references to page numbers, etc. will need to change. Koto documentation produced by users may need to be reformatted with different line leading, using the new fonts.

All of the font files now have a new naming scheme, which allows **FONTSAVAILABLE** to be able to do more accurate pattern matching. For example, the display font file for modern 8 bold italics used to be named:

```
Modern8-B-I-C41.Displayfont
```

The file is now named:

```
Modern08-BIR-C41.Displayfont
```

In general font files use the following format:

The family name (e.g., Modern); a two digit size (e.g., 08); a three letter Face (e.g., BIR, for Bold Italic Regular); the letter C followed by the font's character set in base 8 (e.g., C41); and finally an extension (e.g., Displayfont).

Unknown IMAGEOBJ type
GETFN: SKIO.GETFN.2

The old file naming convention is still supported, however, with the exception of the old Strike file naming convention. In Lyric, **FONTCREATE** will first search for fonts using the new font naming convention, and if the desired font is not found it will search using the Koto convention.

Compatibility considerations You can continue using the old printer metrics (.WD files) in Lyric, thus preserving document looks between Koto and Lyric. If you choose to do so, it is recommended that you rename your old .WD files to the new naming scheme (see above), so that you benefit from the changes to the font searching mechanisms. However, we strongly urge you to use the new .WD files. Otherwise, if you exchange TEdit documents with a site that is using the new files, the documents will print differently at the two sites. The creation date, rather than the naming convention, determines whether a .WD file represents the old or new format.

If, after installing the new .WD files, you wish to print a document using the old Koto formatting, make the font variable **INTERPRESSFONTDIRECTORIES** point to a directory containing the Koto font files. Also any Lyric printer font file information must be uncached from the sysout. To uncache the fonts, perform

```
(for INFO in (FONTSAVAILABLE '* '* '* '* '*
                    'INTERPRESS)
  do (APPLY 'SETFONTDESCRIPTOR INFO))
```

(III:27.30)

(STRINGWIDTH STR FONT FLG RDTBL)

[Function]

In Lyric **STRINGWIDTH** observes ***PRINT-LEVEL*** and ***PRINT-LENGTH***.

In Medley, **STRINGWIDTH** with a NIL argument no longer returns the string width of the string with ***STANDARD-OUTPUT*** font. It now uses **DEFAULTFONT**.

Some new font manipulation functions have been added to Lisp. They are:

(WRITESTRIKEFONTFILE FONT CHARSET FILENAME)

[Function]

Takes a display font font descriptor and a character set number, and writes that character set into a file suitable for reading in again. Note that the font descriptor's current state is used (which was perhaps modified by **INSPECT**ing the datum), so this provides a mechanism for creating/modifying new fonts.

For example:

```
(WRITESTRIKEFONTFILE (FONTCREATE 'GACHA 10) 0
  '{DSK}Magic10-MRR-C0.DISPLAYFONT)
```

writes a font file which is identical in appearance to the current state of Gacha 10 charset 0.

If your **DISPLAYFONTDIRECTORIES** includes {DSK}, then a subsequent **(FONTCREATE 'MAGIC 10)** will create a new font

descriptor whose appearance is the same as the old Gacha font descriptor.

However, the new font is identical to the old one in appearance only. The individual datatype fields and bitmap may not be the same as those in the old font descriptor, due to peculiarities of different font file formats.

Section 27.13 Font Files and Font Directories

(III:27.31)

Press fonts are not part of the sysout since PRESS is now a Library module.

Section 27.14 Font Classes

(III:27.32-27.48)

This section has been expunged from the *InterLisp-D Reference Manual*. Renummer the sections which followed the old Section 27.14 as

SECTION 27.15 ⇒ SECTION 27.14 Font Profiles

SECTION 27.16 ⇒ SECTION 27.15 Image Objects

SECTION 27.17 ⇒ SECTION 27.16 Implementation of Image Streams

Section 27.14 Font Profiles

(III:27.34)

The variable **FONTCHANGEFLG** has an additional value, **ALL**. **FONTCHANGEFLG=ALL** indicates that all calls to **CHANGEFONT** are executed.

(III:27.33-34)

The function **FONTNAME** no longer exists. This function was previously used in Interlisp-D to collect the names and values of variables on **FONTDEFSVARS**. The variable **FONTDEFSVARS** is no longer used; it was appropriate when most output devices were fixed-pitch, "line-printer" style devices, but is not suitable for use when most output devices are laser printers.

Chapter 28 Windows and Menus

Section 28.4 Windows

(III:28.13, 28.38)

The **ADDMENU** function will change a window's **RESHAPEFN** and also will change the window's **REPAINTFN**.

Section 28.4.5 Reshaping Windows

(III:28.17)

The Lisp window system allows the following minimum window sizes:

When creating a new window, the width and height specified must be at least 9, or else you will get an error "region too small to use as a window"

When reshaping a window, the smallest shape you can get is width = 26 and height = height of the font to be used in the window. If you specify a smaller region, **SHAPEW** will simply adjust it to fit these limits.

Section 28.4.8 Shrinking Windows Into Icons

(III:28.22)**SHRINKFN****[Window property]**

In previous releases, there was a bug in the attached window system such that if an attached window had a **SHRINKFN** of the single symbol DON'T, attempting to shrink the window resulted in a break with the message "UNDEFINED FUNCTION DON'T." For this case in Lyric, all windows that can be shrunk will be, while those windows with a **SHRINKFN** of the symbol DON'T will be left open.

To facilitate the management of window regions, the window property **EXPANDREGIONFN** has been added to Lisp. This feature allows applications to arrange for reshaping a window when it is expanded.

EXPANDREGIONFN**[Window property]**

EXPANDREGIONFN, if non-NIL, should be the function to be called (with the window as its argument) before the window is actually expanded.

The **EXPANDREGIONFN** must return **NIL** or a valid region, and must not do any window operations (e.g., redisplaying). If **NIL** is returned, the window is expanded normally, as if the **EXPANDREGIONFN** had not existed. The region returned specifies the new region for the main window only, not for the group including any of its attached windows. The window will be opened in its new shape, and any attached windows will be repositioned or rejustified appropriately. The main window must have a **REPAINTFN** which can repaint the entire window under these conditions.

As with expanding windows normally, the **OPENFN** for the main window is not called.

Also, the window is reshaped without checking for a special shape function (e.g., a **DOSHAPEFN**).

(III:28.23)

Add the variable **DEFAULTICONFN** to the Icon section of the *InterLisp-D Reference Manual*:

DEFAULTICONFN

[Variable]

Changes how an icon is created when a window having no **ICONFN** is shrunk or when **SHRINKW**, with a *TOWHAT* argument of a string, is called. The value of **DEFAULTICONFN** is a function of two arguments (window text); text is either **NIL** or a string. **DEFAULTICONFN** returns an icon window.

The initial value of **DEFAULTICONFN** is **MAKETITLEBARICON**. It creates a window that is a title bar only; the title is either the text argument, the window's title, or "Icon made <date>" for titleless windows. **MAKETITLEBARICON** places the title bar at some corner of the main window.

An alternative behavior is available by setting **DEFAULTICONFN** to be **TEXTICON**. **TEXTICON** creates a titled icon window from the text or window's title. It is described further in Appendix B (ICONW).

(III:28.23)

You can now copy-select titled icons such as those used by FileBrowser, SEdit, TEdit, Sketch. The default behavior is that the icon's title is unread (via **BKSYSBUF**), but if the icon window has a **COPYFN** property, that gets called instead, with the icon window as its argument. For example, if the name displayed in an icon is really a symbol, and you want copy selection to cause the name to be unread correctly with respect to the package and read table of the exec you are copying into, you could put the following **COPYFN** property on the icon window:

```
(lambda (window)
  (il:bksysbuf <fetch symbolic name from window> t))
```

Section 28.4.11 Terminal I/O and Page Holding**(III:28.29)**

TTYDISPLAYSTREAM has been fixed so that it can be successfully used with non-windows.

Section 28.5 Menus

Two features have been added to this section, **ICONW** for creating icons, and **FREE MENU**, for creating and using free menus. Both features were formerly part of the Lisp Library.

The description for **ICONW** is in Appendix C. The **FREE MENU** description is in Appendix D.

The Lyric version of Free Menu differs in some respects from the Koto version of Free Menu. Following is a description of the incompatible feature changes from the old version to the new version of Free Menu. Some of the terminology used in these notes is introduced in the Free Menu documentation found in Appendix B. Please reference Appendix B before reading the following notes.

- The function **FREEMENU** is used to create a Free Menu, replacing and combining the functions **FM.MAKEMENU** and **FM.FORMATMENU**.

The description of Free Menu has these changes:

1. There is no longer a **WINDOWPROPS** list in the Free Menu Description. Instead, the window properties **TITLE** and **BORDER** that were previously set in the **WINDOWPROPS** list can now be passed to the function **FREEMENU**. Other window properties (like **FM.PROMPTWINDOW**) can be set directly after Free Menu returns the window using the system function **WINDOWPROP**. See Appendix B, Section 28.7.14, Free Menu Window Properties.
2. Setting the initial state of an item is now done with the item property **INITSTATE** in the item description, rather than the **STATE** property.

Free Menu Items has been modified as follows:

1. **3STATE** items now have states **OFF**, **NIL**, and **T** (instead of a **NEUTRAL** state). They appear by default in the **NIL** state.
2. **STATE** items are general purpose items which maintain state, and replace the functionality of **NCHOOSE** items. To get the functionality of **NCHOOSE** items, specify the property **MENUITEMS** (a list of items to go in a popup menu), which instructs the **STATE** item to popup the menu when it is selected. **STATE** items do not display their current state by default, like **NCHOOSE** items used to. Instead, if you want the state displayed in the Free Menu, you have to link the **STATE** item to a **DISPLAY** item using a Free Menu Item Link named "DISPLAY". The current state of the **STATE** item will then automatically be displayed in the specified **DISPLAY** item. The item properties **MENUFONT** and **MENUTITLE** also apply to the popup menu.
3. **NWAY** items are declared slightly differently. There is now the notion of an NWay Collection, which is a collection of items acting as a single nway item. The Collection is declared by specifying any number of NWay items, each with the same **COLLECTION** property. NWay Collections have properties themselves, accessible by the macro **FM.NWAYPROPS**. These properties can be specified in property list format as the value of the **NWAYPROPS** Item Property of the first NWay item declared for each Collection. NWay Collections by default cannot be deselected (a state in which no item selected). Setting the Collection property **DESELECT** to any non-nil value changes this behavior. The state of the NWay Collection is maintained in its **STATE** property.
4. **EDIT** items no longer will stop at the edge of the window. Editing is either restricted by the **MAXWIDTH** property, or else it is not restricted at all. The **EDITSTOP** property is obsolete. When you start editing with the right mouse button the item is first cleared.

5. **EDITSTART** items now specify their associated edit item (there can only be one, now) by a Free Menu Item Link named "EDIT" from the **EDITSTART** item to the **EDIT** item.
6. **TITLE** items are replaced by **DISPLAY** items, which work the same way.

With Free Menu, the item interface functions can take the actual item datatype, the item's *ID* or *LABEL*, or a list of the form (**GROUPID** *ITEMID*) specifying a particular item in a group, as the *ITEM* argument.

The description for **ICONW** is in Appendix B. The **FREE MENU** description is in Appendix C.

These changes have occurred in the Free Menu Interface functions:

(FREEMENU DESCRIPTION TITLE BACKGROUND BORDER) [Function]

Replaces **FM.MAKEMENU** and **FM.FORMATMENU**. The desired format is not specified as the value of the **FORMAT** property in the group's PROPS list.

(FM.GETITEM ID GROUP WINDOW) [Function]

Replaces **FM.ITEMFROMID**.

Searches within *GROUP* for an item whose ID property is *ID*.

ID is matched against the item ID and then the item **LABEL**. If *GROUP* is **NIL**, the entire menu is searched.

(FM.GETSTATE WINDOW) [Function]

Replaces **FM.READSTATE**.

Returns a property list of the selected item in the menu. This list now also includes the NWay Collections and their selected item.

(FM.CHANGELABEL ITEM NEWLABEL WINDOW UPDATEFLG) [Function]

Has a new argument order. Now works by rebuilding the item label from scratch, taking the original specification of **MAXWIDTH** and **MAXHEIGHT** into account. *NEWLABEL* can be an atom, string, or bitmap. If *UPDATEFLG* is set, then the Free Menu Group's regions are recalculated, so that boxed groups will be redisplayed properly.

(FM.CHANGESTATE X NEWSTATE WINDOW) [Function]

Has a new argument order.

X is either an item or an NWay Collection ID. *NEWSTATE* is an appropriate state to the type of item. If an NWay collection, *NEWSTATE* is the actual item to be selected, or **NIL** to deselect. Toggle items take either **T** or **NIL** as *NEWSTATE*, and **3STATE** items take **OFF**, **NIL**, or **T**, and **STATE** items take any atom, string, or bitmap as their new state. For **EDIT** items, *NEWSTATE* is the new label, and **FM.CHANGELABEL** is called to change the label of the **EDIT** item.

(FM.RESETSHAPE WINDOW ALWAYSFLG) [Function]

Replaces **FM.FIXSHAPE**

(FM.HIGHLIGHTITEM ITEM WINDOW) [Function]

Replaces **FM.SHADEITEM** and **FM.SHADEITEMBM**.

FM.HIGHLIGHTITEM will programmatically highlight an item, as specified by its **HIGHLIGHT** property. The highlighting is temporary, and will be undone by a redisplay or scroll. To programmatically shade an item an arbitrary shade, use the new function **FM.SHADE**.

Section 28.6.2 Attached Prompt Windows

(GETPROMPTWINDOW MAINWINDOW #LINES FONT DONTCREATE [Function]

In the Lyric release, the prompt window created by **GETPROMPTWINDOW** is *not* independently closeable, as it was in Koto. That is, selecting **Close** from the right-button window menu in the prompt window is the same as selecting it from the menu of any other window in the group—the entire window group is closed.

Section 28.6.3 Window Operations and Attached Windows

(III:28.51)

Communication of Window Menu Commands between Attached Windows is dependent on the name of function used to implement the window command, e.g., **CLOSEW** implements **CLOSE** (refer to **PASSTOMAINCOMS** documentation under Attached Windows). Consequently, if an application intercepts a window command by changing **WHENSELECTEDFN** for an item in the WindowMenu (for example, to advise the application that a window is being closed), windows may not behave correctly when attached to other windows.

To get around this problem, the Medley release provides the variable **attached-window-command-synonyms**. This variable is an ALIST, where each element is of the form (new-command-function-name . old-command-function-name).

For example, if an application redefines the WindowMenu to call *my-close-window* when **CLOSE** is selected, that application should:

```
(cl:push      '(my-close-window      .      il:closew)
il:*attached-window-command-synonyms*)
```

in order to tell the attached window system that *my-close-window* is a synonym function for **CLOSEW**.

Chapter 29 Hardcopy Facilities

(III:29.3)

The **HARDCOPYW** function now has an additional argument, **HARDCOPYTITLE**, which allows you to change or eliminate the "Window Image" message on IP screen images. Moreover, **HARDCOPYW** function now allows you to print large images occupying more than one page.

(**HARDCOPYW** *WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION
PRINTERTYPE HARDCOPYTITLE*) [Function]

HARDCOPYTITLE is a string specifying a title to print on the page containing the screen image. If *NIL*, the string "Window Image" is used. To omit a title, specify the null string.

Chapter 30 Terminal Input/Output

Section 30.1 Interrupt Characters

(III:30.2)

- | | |
|-----------|--|
| Control-P | The Control-P (PRINTLEVEL) interrupt is no longer supported. The interrupt of that name still exists and is defaultly assigned to Control-P, but has no effect on printing. |
| Control-T | The Control-T interrupt flashes the window belonging to the tty process and prints its status information in the prompt window. This avoids disrupting the user typescript. |

(III:30.3)

(**INTERRUPTCHAR** *CHAR TYP/FORM HARDFLG* —) [Function]

If the argument *TYP/FORM* is a symbol designating a predefined system interrupt (**RESET**, **ERROR**, **BREAK**, etc), and *HARDFLG* is omitted or **NIL**, then the hardness defaults to the standard hardness of the system interrupt (e.g., **MOUSE** for the **ERROR** interrupt).

Section 30.2.3 Line Buffering

(III:30.11-12)

The **BKSYSBUF** function has been changed, for compatibility reasons. The description now reads as follows:

(**BKSYSBUF** *X FLG RDTBL*) [Function]

BKSYSBUF appends the **PRIN1**-name of *X* to the system input buffer. The effect is the same as though the user had typed *X*. Returns *X*.

If *FLG* is *T*, then the **PRIN2**-name of *X* is used, computed with respect to the readtable *RDTBL*. If *RDTBL* is **NIL** or omitted, the current readtable of the TTY process (which is to receive the characters) is used. Use this for copy selection functions that want their output to be a readable expression in an Exec.

Note that if you are typing at the same time as the **BKSYSBUF** is being performed, the relative order of the typein and the characters of *X* is unpredictable.

(III:30.12)

Add the function **BKSYSCHARCODE** used in line buffering:

(**BKSYSCHARCODE** *CODE*) [Function]

This function appends the character code *CODE* to the system input buffer. The function **BKSYSBUF** is implemented by repeated calls to **BKSYSCHARCODE**.

Section 30.4.1 Changing the Cursor Image

(III:30.14)

The **CURSOR** record has been changed to a DATATYPE, and its field names have changed in the following way:

Old Field Name	New Field Name
CURSORBITMAP	CUIMAGE
CURSORHOTSPOTX	CUHOTSPOTX
CURSORHOTSPOTY	CUHOTSPOTY

The **CURSORHOTSPOT** field no longer exists; its value can be fetched by composing **CUHOTSPOTX** and **CUHOTSPOTY** into a **POSITION**, or stored by deconstructing a **POSITION** into those fields.

In Lyric, the **CURSORCREATE** function accepted as its argument bitmaps of any size, but caused an obscure error. In Medley, a bitmap that is bigger than 16 high or 16 wide will cause an **ILLEGAL ARGUMENT** error.

Section 30.5 Keyboard Interpretation

(III:30. 19–20)

<u>(KEYDOWNP KEYNAME)</u>	[Function]
<u>(KEYACTION KEYNAME ACTIONS —)</u>	[Function]

KEYNAME is interpreted differently in Lyric: If *KEYNAME* is a small integer, it is taken to be the *internal* key number. Otherwise, it is taken to be the name of the key. This means, for example, that the name of the "6" key is not the number 6. Instead, spelled-out names for all the digit keys have been assigned. The "6" key is named SIX. It happens that the key number of the "6" key is 2. Therefore, the following two forms are equivalent:

```
(KEYDOWNP 'SIX)
(KEYDOWNP 2)
```

Note: The key labeled HELP on the 1186 is named DBK-HELP for use in KEYACTION.

Section 30.6 Display Screen

(III:30.22-23)

<u>(CHANGEBACKGROUND SHADE —)</u>	[Function]
-----------------------------------	------------

The function **CHANGEBACKGROUND** treats the *SHADE* argument as a 4 X 4 texture. The **CHANGEBACKGROUND BORDER** function, on the other hand, treats the *SHADE* argument as a 2 X 8 texture.

Therefore, note that the same *SHADE* argument, when used by the two functions, will not necessarily produce the same background and border shades on the display screen.

(III:30.23)

The **VIDEORATE** function works only on the 1108. Append the following note to the **VIDEORATE** function description:

(VIDEORATE TYPE)

[Function]

Note: **VIDEORATE** does not work on the 1186.

Section 30.7 Miscellaneous Terminal I/O

(III:30.24)

(BEEPON FREQ)

[Function]

The argument *FREQ* is measured in hertz, not in **TICKS**.

Chapter 31 Ethernet

Section 31.3.1 Name and Address Conventions

(III:31.8-9)

Amend the first paragraph, describing **NSADDRESS**, to list, in order, the components of **NSADDRESS**:

Addresses of hosts in the NS world consist of three parts, a network number, a machine number, and a socket number. These three parts are embodied in the Interlisp-D data type **NSADDRESS**. The components of **NSADDRESS** are 32-bit network, 48-bit host, 16-bit socket.

Move the following sentence from page 31.9 of the *IRM* to the last paragraph of Name and Address Conventions on page 31.8:

If you wish to manipulate **NSADDRESS** and **NSNAME** objects directly you should load the Lisp Library Module **ETHERRECORDS**.

NS Address Format

In Medley, you can now specify NS addresses in decimal notation, the form presented by the Chat interface of Network Services software. In this notation, a decimal number is broken up by hyphens every 3 digits, much like commas in standard American numerical notation. You can also specify a full 48-bit host number in octal without breaking it into 16-bit segments.

An NS address is specified in the form:

net#host#socket

If the address contains a hyphen in any field, the entire address is interpreted in decimal; otherwise in octal. The field *socket* is optional, and is defaulted appropriately for the application; if

specified, it is a single integer in the same radix as the rest of the address. The field *net* and its terminating # are optional, defaulting usually to the directly-connected network. The fields *net* and *host* are non-negative integers written in one of three forms:

- A sequence of 16-bit octal numbers, separated by periods.
- A single integer in octal radix.
- A sequence of 3-digit decimal numbers, separated by hyphens.

The special variable ***NSADDRESS-FORMAT*** specifies the form used whenever the system prints an NS address object. Its possible values are:

NIL Octal radix, with the host number in three 16-bit parts, the same as in Lyric.

:OCTAL Octal radix without separators.

:DECIMAL Decimal radix with hyphens.

For example, the following all represent the same address, in the three formats listed above:

1750#0.125000.76771#

1750#25200076771#

1-000#2-852-158-969#

The following functions exist for manipulating NS addresses:

(PARSE-NSADDRESS STR DEFAULTSOCKET) [Function]

Parses the string *STR* into an NS address by the rules listed above, or returns NIL if *STR* is not a well-formed address. If *DEFAULTSOCKET* is non-NIL and the string does not include a socket field, the socket of the resulting NS address is set to *DEFAULTSOCKET*.

(COERCE-TO-NSADDRESS HOST DEFAULTSOCKET) [Function]

Returns an NSADDRESS object corresponding to *HOST*, or NIL if it can't. This function should be called by any software wanting to convert a user-supplied NS host specification into a network address. *HOST* can be any one of the following:

- The name of a host, whose address is found by consulting the Clearinghouse data base.
- A symbol or string in the syntax of an NS address, as described above.
- An NSADDRESS object.
- A list of the form (NSHOSTNUMBER *a b c*), specifying the host number as three 16-bit values. In this case, the network number is omitted (zero).

If *DEFAULTSOCKET* is non-NIL and the socket is unspecified, the socket of the result is set to *DEFAULTSOCKET* (if *HOST* is an NSADDRESS object, it is copied in this case).

Network Routing Maintenance

The representation of Pup and NS routing tables has changed, and the background gateway listener processes have been tuned to significantly reduce their overhead.

The INFO command for the Pup and NS gateway listener processes in the process status window now display their routing tables. Clicking with the left button displays the table in random order, middle button displays the table sorted by network number (this takes a little longer).

Section 31.3.2 Clearinghouse Functions

(III:31.9)

The variable **AUTHENTICATION.NET.HINT** has been added to Clearinghouse Functions. It follows the **CH.NET.HINT** variable in the *Interlisp-D Reference Manual*.

AUTHENTICATION.NET.HINT

[Variable]

AUTHENTICATION.NET.HINT can be set to **CH.NET.HINT** to speed up the initial authentication connection. Its value is interpreted in the same manner as **CH.NET.HINT**.

Section 31.3.3 NS Printing

(III:31.12)

With the Medley release there is now a single Printer Watcher process for all NS printers. This mean you won't get a stack overflow if you hardcopy many files in quick succession.

Section 31.3.5.3 Performing Courier Transactions

(III:31.20-21)

The **COURIER.OPEN** function requires that a courier server be running on the host machine.

Section 31.3.5.3.3 Using Bulk Data Transfer

(III:31.24-25)

The following is a correction and clarification to the description in the *Interlisp-D Reference Manual* of receiving values from a bulk data transfer:

It is possible for a Courier procedure to return both bulk data, in the form of a bulk data sink, and a single value (or list of values) as the normal result of the call. However, the Lisp function **COURIER.CALL** only returns one value, either the bulk data stream (when the bulk data sink argument is NIL) or the regular value. There are two principal ways in which a caller can obtain both values.

The usual way to get both values is to pass a function as the bulk data argument, have it retrieve the bulk data and process it as a side-effect (e.g., store it into a variable bound around the **COURIER.CALL**), then return NIL so that the procedure's returned value is returned from **COURIER.CALL**.

The other way, which is documented incorrectly in the *IRM*, is to pass NIL as the bulkdata argument, thus getting the bulk data stream back from **COURIER.CALL**, process the stream, and then get the procedure's returned value by closing the stream. Contrary to the *IRM*, however, you have to close the bulk data stream using its internal close function, **SPP.CLOSE**, rather than the user-level function **CLOSEF**, which consumes the value internally and returns only the stream.

Section 31.5 Pup Level One Functions

\10MBTYPE.PUP	[Variable]
\10MBTYPE.3TO10	[Variable]

The values of these variables are the 10MB Ethernet encapsulation types for PUP packets and Pup-to-10MB address translation packets, respectively. The initial values of these variables are 512 and 513, respectively. However, these values are illegal for an Ethernet conforming to IEEE 802.3 specifications.

New encapsulation types have been defined for IEEE 802.3 networks. To use them, set the variable **\10MBTYPE.PUP** to 2560 (decimal) and **\10MBTYPE.3TO10** to 2561. Then call either **(RESTART.ETHER)** or **(LOGOUT)**, so that the Ethernet code can reinitialize itself. It may be convenient for a site to smash these values directly into the standard sysout everyone fetches by using the function **READSYS** and its ^V command from the TeleRaid Library module (the sysout must be on disk or a random-access file server). Note that *all* pup hosts on a network (servers as well as workstations) must simultaneously choose to use the new values; those using different values will be unable to communicate with each other. The System Tool must also be upgraded at the same time.

Section 31.6.1 Creating and Managing XIPs

The function NSNET.DISTANCE was previously undocumented.

(NSNET.DISTANCE NET#)	[Function]
------------------------------	------------

Returns the "hop count" to network *NET#*, i.e., the number of gateways through which an XIP must pass to reach *NET#*, according to the best routing information known at this point. The local (directly-connected) network is considered to be zero hops away. Current convention is that an inaccessible network is 16 hops away. **NSNET.DISTANCE** may need to wait to obtain routing information from an Internetwork Router if *NET#* is not currently in its routing cache.

[This page intentionally blank]

This section contains release notes indicating changes that have occurred in the library modules since the Lyric release. Medley changes are indicated with revision bars in the right margin.

Refer to the *Lisp Library Modules* manual, Medley release, for complete documentation of the library modules.

Modules that are New, Moved, or Replaced

Modules Moved from the Library to LispUsers

Big
BitMapFns
BusExtender
BusMaster
CircIPrint
CheckSet
CompileBang
Color
C150Stream
DECL
DInfo
FileCache
HelpSys
Iris
LambdaTran
PCallStats
ReadAIS

Modules Moved from LispUsers to the Library

Cash-File
Hash-File
SysEdit
TableBrowser

Modules Moved to Their Own Manuals

TEdit
Sketch
CML, CMLArray, CMLArrayInspector (part of Xerox Common Lisp)

Modules Moved From the Sysout Into the Library

DEdit
Masterscope
Match
Press

Modules Moved From the Library Into the Sysout

IconW
FreeMenu

Modules Replaced

Old: FX-80stream, FastFX-80stream, FXprinter
New: FX-80Printer

Old: WhereIs
New: Where-Is

New Modules

SysEdit
TableBrowser
TextModules

Details of Changes

4045XLPStream

Enabled its graphics capabilities; added 1108 cable/connector pin-outs.

A new function has been added to allow owners of the international 4045 (non-USA model) to use the 4045XLPStream software.

(4045XLP.CHANGE.MODE *MODE*) [Function]

This function changes the internal parameters of the software to allow printing on A4 paper with the international fonts. *MODE* is a string, either "USA" or "INTERNATIONAL", with the default being "USA". Do not use this function unless you have the international font set and A4 paper tray on a non-USA 4045. A4 page size is 2475 pixels wide by 3525 pixels high in portrait, and 3525 x 2475 in landscape mode.

Cash-File

The new library module Cash-File was formerly in LispUsers. Cash-File is a front end to Hash-File which uses a hash table to cache accesses to hash files. This can provide a significant performance improvement in applications which access a small number of keys repeatedly. For example, the Where-Is library module uses this module to achieve acceptable interactive performance.

Centronics

Added cable/connector pin-out.

Chat

Added information about EMACS.

CopyFiles

When told to copy to a non-existent NS subdirectory, it now asks if it should create it.

DataBaseFns

Clarifications in the documentation of LOADDBFLG and SAVEDBFLG are included in Medley.

EditBitMap

Added a description of its user interface.

FileBrowser

Added enhanced features to Load, Compile, Edit; it now preserves path name of source files when copying to another machine or user; sorts files by attributes; and prints hard copies of directory listings.

The FB command now ignores the package of the attributes you optionally specify, so you can easily use it from a non-Interlisp exec.

The enclosing *'s are now included with the names of the variables *EDITMODE* and *DEFAULT-CLEANUP-COMPILER*.

In addition to having outstanding problems fixed, FileBrowser has several new features and NS enhancements.

New features:

- There is an Abort button available during any operation of indefinite duration.
- You can scroll or reshape a FileBrowser that is "busy", e.g, while doing a Recompute.
- The browser title includes a timestamp of when the browser contents were last Recomputed.
- There is a new subcommand of See, "FileBrowse", which opens a FileBrowser on the selected subdirectory. This replaces the odd functionality of the old See/Edit commands that assumed that any file with null name and extension must be a directory; those commands now always treat the selection as a file. FileBrowse is mainly useful in the following situations:

- When browsing NS directories with depth set finite, or when browsing the top level of a server, which is automatically depth 1.
- When browsing on Unix, a device that gives Lisp no indication of whether a filename is a directory or not.
- There is a subcommand of Recompute, "Set Depth" that can be used to set the enumeration depth for future recomputes and recursive FileBrowses. You can also set the depth in an FB command by appending :DEPTH n to the command line, e.g., FB "{Pogo:}<Carstairs>" :DEPTH 1.

The depth counts levels of directories below the last directory in the pattern not containing a wildcard; depth 1 means just the immediate descendants of that directory. Depth is ignored for nontrivial patterns, i.e., anything but "*.*".

- Another new subcommand of Recompute, "Shape to Fit", widens or narrows the browser so that all fields, and no more, are visible but not wider than the screen.
- Directory items are now displayed like files, e.g., you'll see a single line

Lisp>

rather than the double line

<Carstairs>Lisp>

NIL

In addition, the "page" count for a directory item is now the total page size of the directory subtree rooted there.

- FileBrowser consumes somewhat less storage now, and there have been some performance improvements, especially for very large browsers.

FTPServer

FTPServer now supports DELFILE.

The Medley release fixes several bugs in Lisp's handling of PUP FTP connections relating to password handling and filename recognition.

FX-80Driver

New software, new text, and 1108/1186 cable/connector pin-outs have been added.

Comments are now printed in a compressed font .

GCHax

Documentation contains a new description of the STORAGE function.

Grapher

Grapher can now print graphs larger than one page. The variable GRAPH/HARDCOPY/FORMAT is used to control the format of the graph when printing to paper. See the function HARDCOPYGRAPH and the variable GRAPH/HARDCOPY/FORMAT in the documentation for Grapher for more information.

A new GRAPH.PROPS field has been added to Graph record, which produces a list in property-list format, and is accessed by the function GRAPHERPROP.

Hash

Hash is provided for backwards compatability. New applications should use the Medley library module Hash-File instead of this module.

Hash-File

Hash-File is a new library module, upgraded from LispUsers. Hash-File is similar to but not compatible with the Lyric library module, Hash. Hash-File is modeled after the Common Lisp hash table facility, and Hash was modeled after the Interlisp hash array facility.

Kermit

Reference to an excellent text/reference book has been added.

MasterScope

Break when graying a browser has been fixed.

In Medley, MasterScope .LCOM files have been changed to .dfasl file extensions. MasterScope now recognizes Common Lisp structures.

NSMaintain

The module NSMaintain has been completely revised and has all new documentation. Most commands auto-complete on one or two keystrokes. The Change Password command works again, and there are several new commands for listing objects in the Clearinghouse data base and for manipulating the access lists of groups. There is a more rational set of default inputs offered for most commands, and better feedback is given as to whether a command succeeded or failed.

RS232

The RS232.TRACE function is now documented in the Medley release.

Spy

This version of SPY library module works better with Common Lisp and incorporates several new features:

- Enters the pending mode when you bring up the SPY menu by pressing the left or middle button while the control key is down. Any action invoked from the menu is deferred until you next press the left or middle mouse button. For example, you can delete several nodes and then do one update.
- Keeps track of non-symbol frame names.
- Shows the package prefix of symbols in the display.
- Invokes "Merge" menu item from a node menu allowing for a node to merge with its caller.
- Updates SPY.NOMERGEFNS to correspond more closely to "system" functions in Medley.
- Knows about the Medley interpreter.

TableBrowser

- New functions TB.UNSELECT.ITEM and TB.UNSELECT.ALL.ITEMS fill an inadvertant void in the Lyric version.
- Several off-by-ones in the display algorithms have been fixed.
- Performance on large browsers is improved.
- Clarification of TBAFTERCLOSEFN documentation is included in the Medley release.
- New options to TB.MAKE.BROWSER:
 - The option LINESPERITEM, previously documented but not implemented, is now supported. Alternatively, you can specify explicitly the height of items by giving the options ITEMHEIGHT (total height of each item) and/or BASELINE (the height of the "baseline" relative to the bottom of the item; zero if you don't set it). The BASELINE is used for two things: (1) the ypos of the window is set there when the browser's print function is called, and (2) selection and deletion marks are centered between the baseline and the top of the item. Specifying LINESPERITEM is a shorthand method for setting ITEMHEIGHT to fontheight*#lines and BASELINE to fontheight*(#lines-1)+fontdescent (i.e., font's baseline for the first line of the item), so that the selection marker, deletion lines, and positioning for printing all point at the first line of a multi-line item. One further difference: if you change the font of the browser, TableBrowser will recompute the height and baseline parameters if you specified LINESPERITEM, but not if you specified ITEMHEIGHT.

- You can specify an auxiliary window that is to be horizontally scrolled in parallel with the main window by giving the window as the `HEADINGWINDOW` option. The `WIDTH` of the window's `EXTENT` property is maintained in synch with main window. You still need to create the auxiliary window, attach it where you want it and supply it with a `REPAINTFN`. This is how `FileBrowser` implements its header line consisting of "Name" and the attribute names.
- The option `LINETHICKNESS` specifies how thick to draw deletion lines. It defaults to `TB.DELETEDLINEHEIGHT`, initially 1. Making it the height of an item gives an alternative "total blackout" method of deletion.

TCP-IP

Added revised/expanded installation procedure.

DIR to VMS via TCP now works.

TCP Chat hosts can now be lowercase.

(`TCPFTP.SERVER`) now spawns process and runs in it .

TCP-IP to a Sun returns the top-level directory.

`TCPFTP.DEFAULT.FILETYPES` now contains correct entries for `LCOM`, `lcom`, `DFASL`, and `dfasl`.

Files loaded by the high-level modules `TCPFTP`, `TCPFTP.SRV`, `TCPCHAT`, and `TCPTFTP` automatically load their dependencies. If you load files by hand, you must also load their dependencies first. See the section "File Dependencies," or the TCP-IP documentation for more information.

There is a new flag:

`TCP.ALWAYS.READ.HOSTS.FILE`

[Variable]

This flag is initially T. Setting it to NIL will cause the system to parse the `hosts.txt` file only when the filename (stored in the configuration file) is different from the previously read filename, or the write date of the file has changed. The `hosts.txt` file will always be read at least once when loading the software into a clean `sysout`.

If you change your `IP.INIT` file while TCP-IP is running, you will be prompted to confirm **Restarting TCP**. In most cases, you should confirm the restart.

TExec

A TEXEC executive window no longer has GET in the menu of possible actions, since GETting text into an executive window makes no sense.

TextModules

TextModules is a new library module with the Medley release. It can be used to import and export textfiles and File Manager files. It can bring portable Common Lisp sources into the File Manager without losing any of their contents, and create new textfiles based on the File Manager's description of the textfile contents.

Virtual Keyboards

The Standard-Russian virtual keyboard now has uppercase Be (. . .) and Ve (. . .) in the right places.

Loading VirtualKeyboards now adds the item KEYBOARD to the default window menu as well as the background menu. Selecting this item from the default window menu allows you to specify a keyboard for an individual window.

Where-Is

Where-Is is a new library module, upgraded from LispUsers. This module replaces the Lyric library module WhereIs. This is a new implementation of a facility similar to but not compatible with the Lyric library module WhereIs. Where-Is indexes all definers, but WhereIs only indexed Interlisp FNS definitions.

Additional Notes

DEdit is not error-protected. Doing a ^in a DEdit break window closes the DEdit window, too.

In addition, the modules work under all Lisp environments (Interlisp-D, Xerox Common Lisp, Common Lisp). However, many of the functions and variables used within the modules are those of Interlisp-D, and therefore you'll have to make sure that, when you are not in Interlisp, you use the IL: prefix.

Koto CML Library Module

If you have files that used the Koto CML library module, with its package-style symbol naming conventions, you will need to convert them to use the correct symbols in Lyric /Medley. The procedure is briefly as follows: see the *Common Lisp Implementation Notes*,

chapter 11, "Reader compatibility feature" for complete details on this mechanism:

First, set the global variable LITATOM-PACKAGE-CONVERSION-ENABLED to T. Then for each of your files, do

```
(LOAD file 'PROP)
```

```
(MAKEFILE file 'NEW)
```

Afterwards be sure to set the global variable LITATOM-PACKAGE-CONVERSION-ENABLED back to NIL.

[This page intentionally left blank]

A User's Guide to TEdit—Release Notes

For the Medley Release, TEdit has increased the number of expanded characters, added options to the **Put** and **Get** submenus, clarified several options in the Paragraph Looks and Page Layout menus, and added several minor items to the programmer's interface.

Expanded Characters

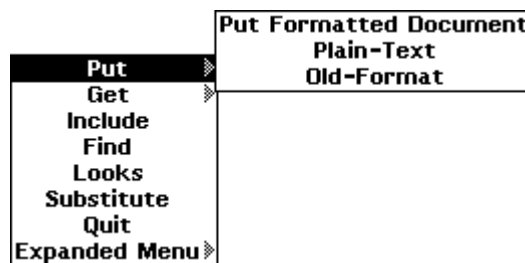
TEdit added the following abbreviations and expansions to the characters shown in Table 1.

Table 1. TEdit's Abbreviations and their Expanded Characters

Abbreviation	Expanded character name	Expansion Character
p	Pilcrow (proofreader's paragraph mark)	¶
t	Trademark	™
tm	Trademark	™
r	Registered trademark	®
1/3	Built-up fraction	⅓
x	Times sign	×
/	Division sign	÷
o (oh)	Degrees sign	°
L	Pound sterling sign	£
Y	Yen sign	¥
+	Plus-or-minus sign	±
^(shift-6)	Up arrow (NS character)	↑
ua	Up arrow (NS character)	↑
	Down arrow (NS character)	↓
da	Down arrow (NS character)	↓
<-	Left arrow (NS character)	←
la	Left arrow (NS character)	←
_ (underscore)	Left arrow (NS character)	←
->	Right arrow (NS character)	→
ra	Right arrow (NS character)	→
=	Two-way arrows (NS characters)	↔

Put Submenu

The drag-through menu for the **Put** command now has the following entries:

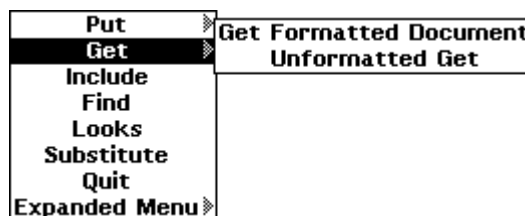


The **Put** command has a submenu that offers you several options for saving your file:

- Keep the formatting in the file. Use this **Put Formatted Document** option, which is the default, unless you have special requirements.
- Save the file as plain text, regardless of formatting. Using this **Plain-Text** option removes all of TEdit's formatting from the file, leaving plain text.
- Save TEdit files in an "old" format. This **Old-Format** option allows you save files in the format of a previous release of TEdit. This format option is provided for backward compatibility.

Get Submenu

The drag-through menu for the **Get** command now has the following entries:



Get has a submenu that offers you the option of retrieving a formatted file (**Get Formatted Document**), or retrieving a file as though it were plain text, with most formatting information appearing as black rectangles (•).

Clarified Paragraph Looks Menu Options

Both the menu options **New Page: Before After** and **Displaymode: Hardcopy** have expanded explanations.

New Page: Before After

Sometimes a page break occurs so that the first paragraph on a page is marked with the **Before** command. In these cases, the text flows continuously from the previous page to this page; a blank page does not appear between them.

Displaymode: Hardcopy

The Hardcopy displaymode command now works only when the text is printed in Interpress fonts.

Clarified Page Layout Menu Options

When specifying text to appear before or after page numbers, you can only enter text in the brackets; image objects are not allowed.

You may now specify a landscape page layout.

In the page layout menu, Modern 10 MRR is now the default page number font instead of Gacha 10. Also, there is a global variable, TEDIT.DEFAULT.FOLIO.LOOKS, that you can set to be any character-looks specification acceptable to TEDIT.LOOKS. The default (i.e., if you don't specify one in the page layout menu) is taken from there.

If you have set page formatting in the past, the page-numbering font has been set as well (even if you specified nothing). This behavior continues, but the default is more sensible, and can be changed.

You may now number the first page of a TEdit file 0 (zero).

TEdit now preserves text before and after page numbers after a file is saved.

Using numbers with decimal points in the "Text before page number" field in the page-layout menu now works properly.

Added Items to Programmer's Interface

The TEXTOBJ data structure has a correction, the TEDIT.INCLUDE, TEDIT.PARALOOKS and TEXTPROP functions are expanded, and the global variable TEDIT.KNOWN.FONTS is now documented.

Corrected the AFTERQUITFN Property

(AFTERQUITFN *WINDOW TEXTSTREAM*) is an optional user-supplied Lisp function that is called after ending an editing session to perform any required cleanup. The *WINDOW* argument was omitted in the manual.

Corrected the TITLEMENUFN Property

TEDIT.TITLEMENUFN is a window property, not a TEdit property as documented in the manual.

Corrected the TEXTOBJ Data Structure

The data structure called TEXTOBJ has as its first field of interest WINDOW, not WINDOW as documented in the manual.

Expanded the TEDIT.INCLUDE Function

TEDIT.INCLUDE now accepts optional START and END arguments that instruct it to restrict its attention to a portion of the TEdit file, treating this part as a separate file. This feature is useful when you require that several distinct TEdit documents reside within a single TEdit file, for example, for database applications. Each document can be formatted and extracted separately.

CAUTION

If you use START and END arguments with INCLUDE, and then format the entire TEdit file, you will lose the formatting.

CAUTION

TEDIT.INCLUDE and OPENTEXTSTREAM take optional arguments that let you take a document out of the middle of a file. This option requires that Lisp be able to determine the length of the file before it is read. Some file protocols (TCP FTP in particular) don't let Lisp do this. If you try to use this option with a file that resides at the other end of a TCP connection (or, more generally, on any device where you cannot tell the length of the file until you have read the whole file), it won't work. The result will be that your document will contain no characters.

Expanded the TEDIT.PARALOOKS Function

TEDIT.PARALOOKS can be used to set NEWPAGEBEFORE, NEWPAGEAFTER, HARDCOPY, TYPE, SUBTYPE, REVISED, KEEP, STYLE, CHARSTYLES, and USERINFO parameters.

REVISED, if non-NIL, causes a revision bar to be printed one pica to the right of the right margin of the paragraph. It is a vertical bar 1 point wide from the top of the top line's ascent to the bottom of the bottom line's descent.

USERINFO can be used as a property list for saving information of interest to the user. It is generally used in a number of undocumented features (e.g. footnote support).

KEEP, STYLE, and CHARSTYLES are reserved for a future release.

Expanded the TEXTPROP Function

You can now also use your own properties, but these properties are not saved with the document if you **Put** it.

Added Documentation for a Global Variable

The following documentation should be added to TEdit's Global Variables.

TEDIT.KNOWN.FONTS

[Variable]

A list of available fonts that appear in the Character Looks menu. The list is in the form ((name-in-the-menu-1 'Real-font-name-1) (name-in-the-menu-2 'Real-font-name-2) ...), for example, ((Classic 'CLASSIC) (Times 'TIMESROMAN)).

Changes to Programmer's Interface to TEdit

STREAM AND TEXTOBJ

All public TEdit functions (non- \) that take a *TEXTOBJ* argument accept either a *TEXTOBJ* or a text *STREAM* as that argument's value.

Changes, Additions and Corrections to TEdit functions

The function TEDIT.SINGLE.PAGEFORMAT is incorrectly documented in the Lisp Library. The following corrections should be noted: The arguments *PG#X*, *PG#Y*, and *PG#FONT* should be *PX*, *PY*, and *PFONT*, respectively.

The argument *PG#ALIGNMENT* should be *PQUAD*.

The order for the arguments, *TOP BOTTOM LEFT RIGHT* should be *LEFT RIGHT TOP BOTTOM*.

The argument *#COLS* should be *COLS*.

INTERCOLSPACE should be *INTERCOL*. And between the *INTERCOL* and *UNITS* arguments there is a *HEADINGS* argument.

The functions and its arguments look like:

```
(TEDIT.SINGLE.PAGEFORMAT    PAGE#S? PX PY PFONT PQUAD LEFT RIGHT
                             TOP BOTTOM COLS COLWIDTH INTERCOL
                             HEADINGS UNITS PAGEPROPS PAPERSIZE)
```

[Function]

PAGE#S? T if you want page numbers on this kind of page, else NIL.

PX The horizontal location of the page number, measured from the left edge of the paper. Negative values are measured from the paper's right edge.

<i>PY</i>	The vertical location of the baseline for the page numbers, measured from the bottom of the paper. Negative values are measured from the top of the paper.
<i>PFONT</i>	The font to be used to display the page numbers. This can be any specification that is acceptable to TEDIT.LOOKS.
<i>PQUAD</i>	An atom that tells how the page number is to be aligned on the location specified by <i>PX</i> and <i>PY</i> . LEFT means the location is the lower-left corner of the page number. RIGHT means the location is the lower-right corner. CENTERED means the page number will be centered around the <i>PX</i> you specified.
<i>LEFT</i>	The left margin—the distance from the left edge of the paper to the left edge of the first text column.
<i>RIGHT</i>	The right margin—the distance from the right edge of the rightmost text column to the right edge of the paper.
<i>TOP</i>	The top margin of the page—the distance from the top of the paper to the top of the first line of body text.
<i>BOTTOM</i>	The bottom margin—the distance from the bottom of the last line of body text to the bottom of the paper.
<i>COLS</i>	Number of columns (default is one).
<i>COLWIDTH</i>	The column width (default is to evenly divide the available space among the #COLS columns).
<i>INTERCOL</i>	The space between the right edge of one column and the left edge of the next column. Defaults to evenly divide the space left after the columns are set up. If there is more than one column, one or the other of <i>COLWIDTH</i> and <i>INTERCOLSPACE</i> must be specified.
<i>HEADINGS</i>	A list of lists in the form of ((<i>HEADINGNAME</i> ₁ <i>XLOCATION</i> ₁ <i>YLOCATION</i> ₁) (<i>HEADINGNAME</i> ₂ <i>XLOCATION</i> ₂ <i>YLOCATION</i> ₂) . . . (<i>HEADINGNAME</i> _n <i>XLOCATION</i> _n <i>YLOCATION</i> _n)).
<i>UNITS</i>	The units used in setting the values you specified. May be one of the atoms PICAS, IN, INCHES, CM, POINTS. Default is POINTS.
<i>PAGEPROPS</i>	<p>A property list of extra information. Properties are STARTINGPAGE#, FOLIOINFO, and LANDSCAPE?.</p> <p>STARTINGPAGE# is the first page's number; it is ignored if this isn't the first page.</p> <p>FOLIOINFO is a list of information about page numbers, (FORMAT TEXTBEFORE TEXTAFTER). FORMAT can be one of ARABIC, LOWERROMAN, UPPERROMAN, or NIL (i.e., ARABIC). TEXTBEFORE is the text preceding the number, and TEXTAFTER is the text following the number.</p> <p>LANDSCAPE? determines if the document is printed in the usual vertical format or printed in landscape format (horizontally). If NIL the document is printed vertically, if non-NIL the document is printed landscape. Defaults to NIL.</p>
<i>PAPERSIZE</i>	Is one of LETTER, LEGAL, the metric paper sizes (A0, A1, A2 A3, A4, A5, B0, B2, B3, B4), or NIL (which defaults to letter size).

TEDIT.GET accepts an open stream as the file to GET from. You may still pass it a TEXTOBJ, however.

(TEDIT.GET *STREAM FILE UNFORMATTED?*)

[Function]

Performs the TEdit Get command, loading the text from *FILE* onto the editing stream *STREAM*—replacing the text that is being edited currently. If *FILE* is not supplied, the user will be asked for a file name. If *UNFORMATTED?* is non-NIL, *FILE* is treated as a plain-text document, and all of its contents are included—even TEdit formatting information.

You can now use TEDIT.PUT to store a TEdit document in the middle of a larger file (e.g., for saving TEdit documents as part of a database). The complete documentation is now as follows:

(TEDIT.PUT *STREAM FILE FORCENEW UNFORMATTED? OLDFORMAT?*)

[Function]

Performs the TEdit Put command, saving the text from the text stream *STREAM* onto the file named *FILE*. If *FILE* is NIL, the user will be prompted for a file name. In this case, if *FORCENEW* is NIL, the user is offered the old file name as a default; if non-NIL, no default is given, forcing the user to specify a file name. If *UNFORMATTED?* is non-NIL, only characters are put in the file—no formatting. If *OLDFORMAT?* is non-NIL, the file will be written in the format used by the previous version of TEdit, for backward compatibility.

In order to store a TEdit document as part of another file, call TEDIT.PUT, passing an open stream on the file as the *FILE* argument. The stream should be open for output and positioned at the place you want TEdit to store the document (call this file pointer *START*). When TEDIT.PUT returns, the stream's end-of-file pointer will be just after the last byte in the newly-inserted document. Call this file pointer *END*. To subsequently retrieve the document from the middle of this other file, call OPENTEXTSTREAM on the file, passing the *START* and *END* pointers as the *START* and *END* arguments.

Note: When TEDIT.PUT returns, the stream will be open for INPUT.

The functions TEDIT.MOVE and TEDIT.COPY were not documented in Koto. They are:

(TEDIT.MOVE *FROM TO*)

[Function]

FROM and *TO* are SELECTIONs. Moves the text described by *FROM* to the place described by *TO*, within the same text stream or between different text streams. The text described by *FROM* is deleted from its original location.

(TEDIT.COPY *FROM TO*)

[Function]

FROM and *TO* are SELECTIONs. Copies the text described by *FROM* to the place described by *TO*, within the same text stream or between different text streams. The text described by *FROM* is not deleted in the *FROM* location.

Changes in the Documentation of TEdit Functions

The following functions have had the documentation of their arguments changed to reflect what will appear if you do a ?= or evaluate ARGLIST on one of these functions. Arguments that were corrected are indicated by bold italics (***arg***). Please note that what changed was the documentation, not the way the functions operate or the values of the arguments themselves.

(TEDIT.SETSEL <i>LEAVECARETLOOKS OPERATION</i>)	STREAM <i>CH#</i> LEN POINT <i>PENDINGDELFLG</i>	[Function]
(COERCETEXTOBJ STREAM TYPE <i>OUTPUTSTREAM</i>)		[Function]
(TEDIT.DELETE STREAM <i>SEL LEN</i>)		[Function]
(TEDIT.INCLUDE <i>STREAM FILE START END</i>)		[Function]
(TEDIT.FIND <i>STREAM TARGETSTRING</i> START# END# WILDCARDS?)		[Function]
(TEDIT.GET.LOOKS <i>STREAM CH#ORCHARLOOKS</i>)		[Function]
(TEDIT.PARALOOKS <i>STREAM NEWLOOKS SEL LEN</i>)		[Function]
(TEDIT.COMPOUND.PAGEFORMAT <i>FIRST VERSO RECTO</i>)		[Function]
(TEXTOBJ <i>STREAM</i>)		[Function]
(TEXTSTREAM <i>STREAM</i>)		[Function]
(TEDIT.CARETLOOKS STREAM <i>LOOKS</i>)		[Function]
(TEDIT.NORMALIZECARET <i>STREAM SEL</i>)		[Function]
(COPYTEXTSTREAM <i>ORIGINAL CROSSCOPY</i>)		[Function]
(TEDIT.PROMPTPRINT <i>TEXTSTREAM MSG CLEAR?</i>)		[Function]
(TEDIT.SETSYNTAX <i>CHAR CLASS TABLE</i>)		[Function]
(TEDIT.GETSYNTAX <i>CH TABLE</i>)		[Function]
(TEDIT.SETFUNCTION CHARCODE FN <i>RTBL</i>)		[Function]
(TEDIT.WORDGET <i>CH TABLE</i>)		[Function]
(TEDIT.WORDSET <i>CHARCODE CLASS TABLE</i>)		[Function]
(TEDIT.INSERT.OBJECT OBJECT STREAM <i>CH#</i>)		[Function]

The following functions were previously documented as accepting a TEXTOBJ. They all still take a TEXTOBJ but they will now also accept a STREAM as the first argument.

(TEDIT.FIND STREAM TARGETSTRING START# END# WILDCARDS?)	[Function]
(TEDIT.GET.LOOKS STREAM CH#ORCHARLOOKS)	[Function]
(TEDIT.PARALOOKS STREAM NEWLOOKS SEL LEN)	[Function]
(TEXTSTREAM STREAM)	[Function]
(TEDIT.NORMALIZECARET STREAM SEL)	[Function]
(TEDIT.PROMPTPRINT TEXTSTREAM MSG CLEAR?)	[Function]

New Features

For the benefit of NS file server users, TEdit now writes files of type TEDIT, instead of BINARY. As a result, LISTFILES and the FileBrowser are able to determine that the file is a TEdit file and call

TEdit to create the hardcopy. Previously, it was necessary that the TEdit file explicitly have the extension ".TEdit".

```
(OPENSTREAM file 'OUTPUT 'NEW '((TYPE TEDIT))).
```

This change is for formatted files only. Plain text files are still written as type TEXT. Also, on devices that don't support arbitrary file types (such as conventional mainframe file servers), the type TEDIT coerces to BINARY. Unfortunately, if you subsequently copy the file to an NS file server from such a device, the knowledge of its "true" file type is lost.

A User's Guide to Sketch—Release Notes

The Medley release of Sketch includes several new features, many added in response to user's requests. A programmer's interface allows sketches to be created by programs. This interface is described in a separate document (*The Programmer's Interface to Sketch*.)

Manipulating Sketch Elements

Adding and Deleting Control Points

Individual control points can now be added to and deleted from wires and curves.

Deleting Control Points

You now have the option to delete elements or delete a control point. Just select the **Delete** command, move the mouse cursor out through the grey arrow, then select the point to be deleted.

Defaults Command

Better Feedback for Creating Wires, Circles and Ellipses

Sketch now provides better feedback when you are creating circles, ellipses and wires. You are now prompted with an image of what the figure will look like if you release the left button. You can get the old feedback behavior (for example, if this is too slow) by selecting the **Feedback** subcommand from the **Defaults** submenu, then selecting the **Points only** subcommand from its submenu.

Arrowheads

A curved arrowhead shape was added and is now the default. Also, a command was added to the menu of arrowhead change operations that implements "look same" for arrowheads. To make the arrowheads on a collection of elements look the same: select **Change**; then, when prompted to select the elements to change, first select the element that has the desired arrowhead, then, in the same selection, add the elements that you want to look like the first one; then select the item **Arrowheads**, then the item **Both**, then the item **Same as First**.

Deleting Characters During Type-in

You can now delete characters by using the UNDO key, just as you would in TEdit. Type in a word or a phrase, then press the UNDO key, and the text will be deleted.

Using Bit Maps in a Sketch

Zooming Bitmaps

The bit image element provides a bitmap that zooms. Selecting the **Bit image** command from the command menu will prompt you for a region of the screen that will be inserted as a bit image into the sketch.

Changing Bitmaps

When you apply a **Change** command to a bit image that it is being viewed at actual size, you will be prompted with the same menu as a bitmap image object. If the image is being displayed at other than original scale, you will be given the menu shown below.

Scaled bitmap operations

Perform edit operations on the source bitmap of this image.
 Make the image shown be the source
 Make the source be at this scale
 Make the image shown be the source at the source scale
 Save this image to be used as a source at this scale

*Menu of commands offered when **Change** command is applied to a bit image that is not at the original scale.*

Freezing Sketch Elements

It is now possible to freeze elements, that is to make them unaffected by edit changes. Frozen elements will not have their control points highlighted (and hence cannot be selected) after an edit command has been selected. This provides a way to keep part of the figure fixed while editing on an overlapping part. It also reduces the number of control points. The **Freeze** command is a subcommand to the **Group** command. It will prompt you for a collection of elements that will then be frozen. Elements can be unfrozen by the **UnFreeze** command that is a subcommand to the **UnGroup** command.

Aligning Sketch Elements

Sketch contains a set of commands to align elements. The main menu command **Align** prompts for a collection of control points and moves them so that they all line up with the leftmost one.

Placing Multiple Copies of Elements

There is a new feature in Sketch that makes it much easier to place multiple copies of a collection of elements. While positioning the image of the elements during the **Copy** command, hold down the COPY key. A new copy of the elements will be positioned everytime a mouse button (left or right) is pressed and released, until either the image is placed completely outside the viewer or the COPY key is released before the mouse button is released.

Making the Window Fit the Sketch

The **Fit to window** subcommand under the **Move View** command will zoom the sketch so that it just fits within the current window. It

has a sub-subcommand **Fit window to sketch** that will reshape the window so that the entire sketch (at the size shown) just fits within it. This is useful if you change a sketch that was edited from a document.

Overlaying Figure Elements

Elements that have a filling property (boxes, text boxes, circles, polygons and closed curves) now have a mode property that determines how the filling should effect elements it covers. The option **Filling mode** now appears in the **Which aspect?** submenu.

Changing How Elements Overlap

Elements have an order in which they are displayed. An element that is displayed early can be covered by elements layed down later. Thus, changing the order in which overlapping elements are displayed can effect the resulting image. The **Bury** command provides three subcommands to change the order in which elements are displayed.

The **Bury** command will prompt you to select an element or elements and will change their order so that they are displayed first. That is, they will appear underneath any other elements. If you select more than one element, they will all be displayed before any non-selected elements and their relative order maintained. The **Send to bottom** subcommand does the same thing as **Bury**.

The **Bring to top** command is a subitem to the **Bury** command. It will prompt you to select an element or elements and will change their order so that they are displayed last. That is, they will appear on top of any other elements. If you select more than one element, they will all be displayed after any non-selected elements and their relative order maintained.

The **Reverse order** command is a subitem to the **Bury** command. It will prompt you to select a collection of elements and will reverse their display orders. A special case is when two elements are selected. In this case the element positions are switched.

Loading the Sketch Library Module in the 1186 Environment

The SKETCH executable files are too large to be contained on one floppy. The files are now distributed on two floppies: *Medley Library Floppy #3* and *Medley Library Floppy #4*. To load SKETCH, type the Interlisp exec command:

(FILESLOAD LOAD-SKETCH)

The LOAD-SKETCH function will copy all SKETCH files from #3; then prompt you to insert #4, and the remainder of the files will be copied.

The Programmer's Interface

The programmer's interface allows Sketch to be used as a tool by other programs. It is documented in the *Programmer's Interface to Sketch*.

New Behavior for the Get Command

The action of the **Get** command was changed to be consistent with the TEdit **Get** command. It now deletes any sketch elements that are in the sketch prior to the **Get** command. The affect of the old **Get** command is available as the **Include** command on a submenu to the **Get** command.

Establishing Initial Defaults for Sketch

The variable SK.DEFAULT.FONT, if non-NIL, is used as the default font. If SK.DEFAULT.FONT is NIL, the default font (DEFAULTFONT) is used.

The following variables are used to establish the default setting for a new sketch. Descriptions of legal values can be found in the *Programmer's Interface to Sketch*. SK.DEFAULT.BRUSH is the default brush. SK.DEFAULT.ARROW.LENGTH is the default arrowhead size. SK.DEFAULT.ARROW.TYPE is the default type (one of LINE, CURVE, CLOSEDLINE or SOLID). SK.DEFAULT.ARROW.ANGLE is the default angle for arrowheads. SK.DEFAULT.TEXT.ALIGNMENT is the default text alignment. SK.DEFAULT.TEXTBOX.ALIGNMENT is the default textbox alignment. SK.DEFAULT.DASHING is the default dashing. SK.DEFAULT.TEXTURE, SK.DEFAULT.BACKCOLOR and SK.DEFAULT.OPERATION are combined to create the default filling.

1108 User's Guide Release Notes

What to Look For

The *1108 User's Guide* was extensively reorganized and rewritten for the Lyric Release. This made it nearly identical to the *1186 User's Guide*. This section contains a summary of changes affecting 1108 environments with the Medley release. Details are described in update pages available for the *1108 User's Guide*.

In every 1108 chapter that requires use of Lisp expressions of any kind, there is a notice regarding the use of **IL:** and a suggestion that expressions, functions, and variables be typed into an Interlisp Exec.

4. File System

Medley will accept floppy names up to 40 characters in length. Some of the Lyric font floppies have names in excess of 40 characters. Medley truncates the floppy name to 40 characters if asked to read a Lyric floppy with a longer name. The function FLOPPY.NAME is used to name a floppy. When it is not given any arguments, it returns the name stored on the floppy disk. When it is given a *NAME* argument, the floppy name is set to *NAME*. The 40 character limitation holds for both 1108 and 1186 floppies.

The {DSK} device on the 1108 and 1186 now accepts a wider range of characters in file names. Almost any character in character set 0 is acceptable. Previously, if you tried to create a file whose name included, for example, an underscore, you would see a "FILE NOT FOUND" error.

The 1108 and 1186 file systems had a problem with large partitions which would manifest itself as "HARD DISK error - can't find file page" when accessing newly created files. This would only appear on logical volumes larger than 64K pages. This problem has been fixed.

The function FILENAMEFROMID is now implemented.

6. System Tools

In System Tools, it is no longer necessary to execute a **Floppy Info!** command before attempting a **List!**.

The Medley System Tool now displays an error message when an NS Domain or Organization name is more than the allowed 20 characters long.

The Medley System Tool now supports sysout and microcode installation using the TCP FTP protocol. This feature may be used by selecting the "TCP/FTP" device type in the main System Tool window. Update pages for the *1108 User's Guide* describing this feature, are included with the Medley release.

7. Input/Output

Every time you allocate space on a floppy disk that has fewer than 200 free pages, a message is printed in the prompt window. That message gives an approximate number of free pages remaining after the allocation; it's intended to give you warning when your floppy is nearing full. The page count is correct only within +/- 2 pages because the message is printed in the course of the allocation, and the floppy's directory may grow when the new file is added to it .

8. Machine Diagnostics

Medley Boot Diagnostics for the 1108 include changed floppy disk names and slight changes in the prompts for running diagnostics from floppies.

1186 User's Guide Release Notes

What to Look For

The *1186 User's Guide* was extensively reorganized and rewritten for the Lyric Release. This section contains a summary of changes affecting 1186 environments with the Medley release. Details are described in update pages available for the *1186 User's Guide*.

In every 1186 chapter that requires use of Lisp expressions of any kind, there is a notice regarding the use of **IL:** and a suggestion that expressions, functions, and variables be typed into an Interlisp Exec.

1. Introduction

For Medley, the Xerox Lisp logo window has been changed to reflect the new name, Envos.

4. File System

Medley will accept floppy names up to 40 characters in length. Some of the Lyric font floppies have names in excess of 40 characters. Medley truncates the floppy name to 40 characters if asked to read a Lyric floppy with a longer name. The function FLOPPY.NAME is used to name a floppy. When it is not given any arguments, it returns the name stored on the floppy disk. When it is given a *NAME* argument, the floppy name is set to *NAME*. The 40 character limitation holds for both 1108 and 1186 floppies.

The {DSK} device on the 1108 and 1186 now accepts a wider range of characters in file names. Almost any character in character set 0 is acceptable. Previously, if you tried to create a file whose name included, for example, an underscore, you would see a "FILE NOT FOUND" error.

The 1108 and 1186 file systems had a problem with large partitions which would manifest itself as "HARD DISK error - can't find file page" when accessing newly created files. This would only appear on logical volumes larger than 64K pages. This problem has been fixed.

The function FILENAMEFROMID is now implemented.

5. Software Installation

The SKETCH executable files are too large to be contained on one floppy. The files are now distributed on two floppies: *Medley Library Floppy #3* and *Medley Library Floppy #4*. To load SKETCH, type the Interlisp exec command:

(FILESLOAD LOAD-SKETCH)

The LOAD-SKETCH function will copy all SKETCH files from #3; then prompt you to insert #4, and the remainder of the files will be copied.

6. System Tools

In System Tools, it is no longer necessary to execute a **Floppy Info!** command before attempting a **List!**.

The Medley System Tool now displays an error message when an NS Domain or Organization name is more than the allowed 20 characters long.

The Medley System Tool now supports sysout and microcode installation using the TCP FTP protocol. This feature may be used by selecting the "TCP/FTP" device type in the main System Tool window. Update pages for the *1186 User's Guide* describing this feature, are included with the Medley release.

7. Input/Output

Every time you allocate space on a floppy disk that has fewer than 200 free pages, a message is printed in the prompt window. That message gives an approximate number of free pages remaining after the allocation; it's intended to give you warning when your floppy is nearing full. The page count is correct only within +/- 2 pages because the message is printed in the course of the allocation, and the floppy's directory may grow when the new file is added to it.

The following function applies only to 1186 users:

(**dove.xor.cursor** &*optional xor-p*)

[Function]

If no argument is given, this function returns the current state of the 1186 cursor (nil implies an or'ing cursor, t an xor'ing cursor). If an argument is given, changes the state of the 1186 cursor appropriately.

8. Diagnostics

Medley Boot Diagnostics for the 1186 include changed floppy disk names and slight changes in the prompts for running diagnostics from floppies.

[This page intentionally left blank]

This section describes new features and enhancements that implement Common Lisp into the Lisp operating environment within the Medley release. This information supplements the *Common Lisp Implementation Notes*, Lyric release. Medley enhancements are indicated with revision bars in the right margin.

New Features Since Lyric

The following description summarizes the new Common Lisp implementation features that have been added or changed since the Lyric release.

New compiler Interface -- The Medley compiler gives better progress reports and it is now possible to invoke the compiler on any definer (not just functions, as before).

New Implementation of Defstruct -- A new version of defstruct compiles more compactly and gives more options so that defstruct has at least as much functionality as the Interlisp record package.

Adoption of features and clarifications suggested by the Common Lisp Cleanup Committee -- Among other changes, the behavior of **append** on dotted lists is now better defined, and a new function **xcl:row-major-aref** has been added.

Common Lisp Veneer on the Interlisp record package -- A collection of macros that make the use of existing Interlisp datatypes more appealing has been added.

Performance enhancements -- A closure caching scheme now insures that repeated calls to symbol-functions of the same symbol will return EQ compiled-function objects.

New opcodes have been added for several common list functions, such as **member** and **assoc**.

Common Lisp Definers

The Medley release contains a new implementation of definers and a reworking of the top level of the XCL Compiler. These represent upward compatible changes that have the effect of allowing the Common Lisp compiler to print out progress reports indicating which definer is currently being compiled. To receive the full benefit of these changes, recompile any file containing a **defdefiner** expression.

It is now possible to compile individual definers by using any of the following forms:

Compile-Definer

(xcl:compile-definer *name type*)

Compile and install the definer of type *type* named *name*.

EXAMPLE:

```
(xcl:compile-definer 'foo 'structures)
```

In this example, the definer will compile and install the structures definition of foo.

Compile-Form

(xcl:compile-form *form*)

Compile and evaluate *form*.

EXAMPLE:

```
(xcl:compile-form '(progn (defconstant c 1) (defun foo (a b) (+ c a b))))
```

In this example, the definer will compile and evaluate the progn using compile-file semantics.

EXAMPLE:

```
(xcl:compile-form '(with-collection (dotimes (i 10) (collect i))))
```

In this example, the definer returns:

```
(0 1 2 3 4 5 6 7 8 9)
```

Define-File-Environment

Rather than establishing **il:makefile-environment** props and **il:filetypes** on the root name of a file, you can define a file environment using the form:

(xcl:define-file-environment *filename* &key *readtable package base compiler*)

This produces an object of file-manager type **xcl:file-environments**. The *filename* can be either a string or a symbol. The rootname of the file is constructed by interning the *filename* in the Interlisp package. Puts the *compiler* argument (if any) under the **il:filetype** prop of the file rootname. Puts the *readtable*, *package* and *base* arguments (if any) under the **il:makefile-environment** prop of the file rootname. None of the arguments are evaluated. There are no defaults.

EXAMPLE:

```
(xcl:define-file-environment myfile :package "XCL-USER" :readtable "XCL" :compiler :compile-file)
```

In this example, *compile-file* is put under the **il:filetype** prop of *myfile*. The *readtable*, *XCL* and *compile* arguments are put under the **il:makefile-environment** prop of *myfile*.

NOTE: **xcl:define-file-environment** is a definer and hence will not be installed if **il:dfnflg** is **il:prop** or if a file is prop loaded.

Site-Name Special Uses

The following special variables are defined and may be set in your init file to inform Common Lisp of site information:

xcl:*short-site-name*

This variable is used in the function **short-site-name**.

xcl:*long-site-name*

This variable is used in the function **long-site-name**.

EXAMPLES:

```
(setq xcl:*short-site-name* "AIS")
(setq xcl:*long-site-name* "Artificial Intelligence Systems")
```

In these examples, (short-site-name) returns "AIS" and (long-site-name) returns "Artificial Intelligence Systems".

Record Access

The Medley release contains several methods for accessing existing Interlisp records using Common Lisp syntax. These features help to integrate Interlisp and Common Lisp. The following sections describe these additions.

Define-Record

(xcl:define-record *name* *interlisp-record-name*

&key *conc-name* *constructor* *predicate* *fast-accessors*) [Definer]

Creates a structures object named by the symbol *name* that provides Common Lisp accessors, setters, predicates and constructors for the Interlisp record named by the symbol *interlisp-record-name*. The Interlisp record must be defined before the **xcl:define-record** expression is evaluated. The keyword arguments are treated as in **defstruct**. The package of constructed names is taken from the value of ***package*** at the time of evaluation (as in **defstruct**). The system contains no predeclared **define-records**.

EXAMPLE:

The form:

```
(xcl:define-record menu il:menu)
```

allows you to write:

```
(menu-items foo) and (setf (menu-items foo) fie)
```

rather than:

```
(il:fetch (il:menu il:items) il:of foo)
```

Record-Fetch

(**xcl:record-fetch** *record field object*)

[Macro]

Evaluates *object*. Does not evaluate *record* and *field*. Both *record* and *field* must be symbols. Symbols with the same p-names are interned in the Interlisp package and are used to construct an **il:fetch** form. **xcl:record-fetch** may be used with **self** and expands to the suitable replace form.

Record-FFetch

(**xcl:record-ffetch** *record field object*)

[Macro]

Similar to **xcl:record-fetch**, but an **il:ffetch** form is generated instead. Evaluates *object*. Does not evaluate *record* and *field*. Both *record* and *field* must be symbols. Symbols with the same p-names are interned in the Interlisp package and are used to construct an **il:ffetch** form. Ffetch may be used with **self** and expands to the suitable freplace form.

Record-Create

(**xcl:record-create** *record &rest keyword-pairs*)

[Macro]

Evaluates the second element of each pair. Does not evaluate *record* (*record* must be a symbol). A symbol with the same p-name is interned in the Interlisp package and used to construct an **il:create** form. The rest of the arguments form keyword pairs. The first element of each pair should be a symbol such that a symbol with the same p-name exists in the Interlisp package and names either a valid slot for this record or is one of **:using**, **:copying**, **:reusing**, or **:smashing**.

Array Reference

(**xcl:row-major-aref** *array index*)

[Function]

Returns the element of *array* given by the row-major-index *index*. The array can be of any dimension. This function can be used with **self**.

Shadowing of Global Macros

The XCL Compiler now properly handles shadowing of global macros by lexical functions. In the Lyric Compiler, lexical functions defined with **flet** did not shadow global definitions of the same name. This has been fixed in Medley.

Evaluating Load-time Expressions

The XCL Compiler now handles **il:loadtimeconstant** correctly. The new Compiler substitutes the entire expression for each reference to the value of a load-time constant. There are potential problems if the code depends on the expression being evaluated exactly once, e.g. if it contains (IDATE).

Common Lisp Defstruct Options

The Medley release contains a new implementation of **defstruct** that offers greater compiled-code compaction, and several new extensions that increase efficiency. This implementation introduces functionality that allows **defstruct** to parallel the Interlisp

record module in flexibility. These features also help to integrate Interlisp and Common Lisp. The following sections describe these additions.

Defstruct Options

:inline

Can be one or both of :accessor and :predicate or t, implying '(:accessor :predicate) or nil, implying no optimizations allowed or :only, implying all accessors and the predicate will be inline only and not funcallable (not usable with the Lisp primitive "funcall"). The default is '(:accessor :predicate).

Copiers and constructors are never inline. The option (:inline :only) implies that no funcallable accessors will be generated (similarly, the predicate, if any, will not be funcallable).

:fast-accessors

Can be t or nil. t implies inline accessors will not type check. The default is nil.

Note that funcallable accessors (if any), always type check, if possible.

NOTE: This represents a change from the Lyric implementation, which allowed specification of a list of slot names that had fast inline accessors.

:template

Can be t or nil, t implies that no datatype will be instantiated. (:template t) implies no :type option. The default is nil.

Templated defstructs have no predicates, copiers or constructs. It is an error to supply any such option in combination with (:template t). Templated defstructs are intended to be used as are IL:blockrecord's. It is possible for a templated defstruct to include another templated structure, but it is an error for a standard defstruct to include a templated structure.

Funcallable accessors (accessors that may be used with the Lisp primitive "funcall") share code with suitable closure templates if the defstruct is compiled with the XCL Compiler. Byte compiled defstructs still generate explicit defun's for all funcallable accessors.

Defstruct Slot Options

:type

The following specialized types are recognized:

(unsigned-byte {1 - 16})

(signed-byte {16, 32})

float, etc.

(member t nil)

il:fullpointer

il:xpointer

il:fullxpointer

Warning When Using Defstruct

Defstruct automatically generates a number of auxilliary functions without checking whether redefining those functions will affect the system. To avoid redefining key functions, you should be aware of the names that will be used. For example:

Do not attempt to define a Structure named TREE. This use of Defstruct implicitly redefines the built-in Common Lisp function COPY-TREE, which renders your system inoperable.

If you have already tried to define a (DEFSTRUCT TREE A B) structure by mistake, you will need to reload your system.

Macros for Collecting Objects

xcl:with-collection

(**xcl:with-collection** &body forms) [Macro]

(**xcl:collect** form) [Macro]

This pair of macros is provided for efficiently collecting objects into a list. In Common Lisp, there is no direct facility provided for doing this, so one must either push objects onto a list, then reverse it, or maintain a tail pointer to the list and use **rplacd** to add new items. The latter has an efficient implementation in Xerox Common Lisp, and **xcl:with-collection** is provided to take advantage of it.

Lexically within the body of an **xcl:with-collection**, the macro **xcl:collect** is defined. It will append the value of its argument to the end of the list being collected. The value of **xcl:with-collection** is the collected list.

xcl:collect may be used inside of functions passed as arguments to other functions.

EXAMPLE:

```
(xcl:with-collection
  (maphash
    #'(lambda (key val)
      (when (interesting-p val) (xcl:collect key)))
    the-hash-table))
```

will collect a list of all the "interesting" keys in the order that they were encountered.

It is an error to use **xcl:collect** outside the scope of an **xcl:with-collection**. Proper lexical nesting is observed, so an instance of **xcl:collect** applies to the most deeply nested **xcl:with-collection** that is found in.

Macros for Writing Macros

xcl:once-only

(xcl:once-only (*{ variable }**) &body *forms*)

[Macro]

This macro is provided to aid in writing macros. **xcl:once-only** helps solve the problem of multiple evaluation of subforms of a macro.

EXAMPLE:

```
(defmacro test (reference form)
  `(setf ,reference (cons ,form ,form)))
```

This example has the problem that **form** will be evaluated twice. To avoid this, one might instead write:

```
(defmacro test (reference form)
  (let ((value (gensym)))
    `(let ((,value ,form))
      (setf ,reference (cons ,value ,value)))))
```

This solves the problem of multiple evaluation, but introduces some others. If **form** is in fact something simple, like a reference to a variable or a literal, there was no need to create the temporary variable, thus "wasting" a symbol. This can be extremely important in Xerox Common Lisp as symbol space is limited and symbols are never reclaimed. If there are many temporary values to be computed, the macro definition becomes cluttered with calls to **gensym** that obscure the essence of the code.

xcl:once-only helps solve these problems. For each of the variables listed, **xcl:once-only** determines if its value (at macroexpansion time) is simple: a symbol or a literal. If it is, appearances of that variable in the macroexpansion will remain unchanged. If it is not, the macroexpansion will contain code to store the value in a temporary **gensym**'ed variable and use that variable in the macroexpansion. Thus, the example could be written as

```
(defmacro test (reference form)
  (xcl:once-only (form)
    `(setf ,reference (cons ,form ,form))))
```

Then `(test (aref the-array x) y)` will expand to something like

```
(setf (aref the-array x) (cons y y))
```

while `(test (aref the-array x) (random-form))` will expand to something like

```
(let ((#:g377 (random-form)))
  (setf (aref the-array x) (cons #:g377 #:g377)))
```

Note that **xcl:once-only** does not attempt to preserve order of evaluation. If this is important then you will still have to create temporary variables yourself.

Common Lisp Append Datatypes

A clarification adopted by X3J13 involves the behavior of the APPEND function with non-lists. The cdr of the last cons in any but the last argument given to APPEND is discarded (whether NIL or not) when preparing the list to be returned. In the case where there is no last cons (i.e., the argument is not a list) in any but the last list argument, the entire argument is effectively ignored. In this situation, if the last argument is a non-list, the result of APPEND can be a non-list. NB: APPEND and COPY-LIST now produce different results for non-lists.

EXAMPLE:

```
(append '(a b c . d) '())
```

produces the result:

```
(a b c)
```

EXAMPLE:

```
(append '(a b . c) '() 3)
```

produces the result:

```
(a b . 3)
```

EXAMPLE:

```
(append 3 17)
```

produces the result:

```
17.
```

Closure Cache

The Medley sysout contains a closure cache that provides increased time and space efficiency. Less new memory is allocated because repeated calls to symbol-function of the same symbol now will cons exactly one closure object. Repeated calls to symbol-function of the same symbol now return EQ-compiled function objects.

Symbols and Packages

Pkg-goto and In-package

PKG-GOTO is now a synonym for IN-PACKAGE. The PKG-GOTO function can be used to change packages in an exec.

PKG-GOTO takes one argument, which can be either a double-quoted string, a symbol, or a package structure. This function is used to set package in an exec.

(xcl:pkg-goto *package-name* &key *nicknames use*) [Function]

PKG-GOTO operates like IN-PACKAGE, but asks for confirmation if a new package is being created. The function is useful at the top level in the exec, to avoid creating new packages when a name is misspelled.

Defpackage Export argument

Defpackage's EXPORT argument now accepts strings. Optionally, strings can be given to :EXPORT instead of symbols. This is recommended when defpackage is used in the makefile-environment property of a file. The strings are interned in the package being defined and then exported.

Debugging Tools

Breaking

Even with HELPDEPTH set to zero, some errors do not cause a break. In Koto and the old Interlisp execs in Lyric, the workaround is:

```
(SETTOPVAL 'HELPFLAG 'BREAK!)
```

In Medley and Lyric's new execs, HELPFLAG is bound but not continually reset. The workaround:

```
(SETQ HELPFLAG 'BREAK!)
```

affects the current exec until the next time you call RESET (or control-D). If you want the change in HELPFLAG to be seen by other processes, you still need to use SETTOPVAL, and RESET any execs in which you want to see the effect.

For related information, see the Medley error system variable XCL:*BREAK-ON-SIGNALS* described in Appendix E.

Advising

In Lyric, putting a second piece of advice on a function caused the system to believe that the function was in fact not advised, so any further advice threw out the already existing advice. This has been fixed. In Medley, the correct list entries are made regardless of whether the function was previously advised.

In Lyric, loading a file with advice caused multiple instances of the advice to be instantiated. To prevent this, ADVISE is now changed in Medley in the following way: When a new piece of advice is put on a function, the system examines the already existing advice to see if the same advice is already there. If so, the old advice is removed before adding the new advice. Sameness is determined by a test similar to CL:EQUALP, except that case distinctions are significant in strings and characters. The priority and location of the advice is taken into account when determining the "sameness." This makes it possible, for instance, to have identical advice be both :FIRST and :LAST.

Advice is no longer replicated when loaded more than once.

The debugger and inspector now display interpreted lexical closures conveniently. Displayed lexical closure contents include the function contained, and any lexical bindings in the closure. Compiled closures are not conveniently inspectable. Common Lisp eval stack frames show their associated lexical environment in a similar manner.

The :when option to XCL:BREAK-FUNCTION no longer causes the broken function to return NIL when the break is not taken. The correct values are returned.

Argument Names Displayed for Interpreted Functions

In the debugger, the frame inspector window will now display the argument names for interpreted Common Lisp functions. Previously, it gave them pseudonames "arg0" "arg1" etc.

Lexical Variables Evaluated by Debugger

The debugger EVAL command now evaluate expressions in the lexical environment --i.e., you can evaluate an expression and use variables that are lexically bound in your code. Only the lexical environment at the point of the break can be evaluated. You can't presently back up to any given lexical environment.

EXAMPLE:

```
(defun fact(x)(if(= 1 x)nil(*x(fact(1-x)))))  
(fact 4)
```

;; breaks. if you then type

```
EVAL x  
2
```

Pathname Component Fixed in FS-ERROR

In Lyric, only one of the three FS-ERROR conditions was passed a pathname component, resulting in the File Cacher not knowing which file had the error, or resulting in pathname being lost when PROTECTION VIOLATION or FILE SYSTEM RESOURCES EXCEEDED were signaled. This problem occurred most noticeably in Lyric when Interlisp errors were translated to XCL. This condition has been fixed in Medley. FS-ERROR now correctly receives all the pathname components.

Compiler Optimizations

Warning when using LABELS construct

In Lyric, use of the LABELS construct generated circular structure that would not get collected. Interpreted, a LABELS construct always creates this non-collectible structure. Compiled, such structure would be created if there were non-tail-recursive or mutually referencing subfunctions. The values of any closed-over variables are captured by this structure and thus also not collected, potentially causing large storage leaks. The latter situation has been relieved somewhat for Medley.

In Medley, the unavoidable circularity has been reduced to include only the mutually referencing functions, but not any of the other data that they access. Thus, the uncollectable structure is created only when a new copy of the code blocks are created, such as by compiling the function containing the LABELS rather than each time that function is called.

COMS added to dfasl files

The Medley compiler has been modified to better handle the il:define-file-info, and defpackage forms. Now, loading a dfasl file

is not implicitly SYSLOAD. Since the file COMS for the file is now included in the dfasl, that file will be noticed by the file manager unless the load is explicitly SYSLOAD. (SYSLOADing of compiled lcom and dfasl files is recommended.)

In Lyric, dfasls of file manager files did not contain the COMS of the file. In Medley, COMS are present in dfasl files, just as they are in lcom files. As with lcom files, the COMS will not be loaded when the LDFLG argument to LOAD is SYSLOAD, nor will the name of the file be added to FILELST, but instead will be added to SYSFILES.

Note: We discourage loading either sort of compiled file (lcom or dfasl) with any value for LDFLG but SYSLOAD. Unless you intend to edit a file, you should always load it SYSLOAD. Even when you intend to edit it, it is usually preferable to SYSLOAD it and then load the source PROP. If there are too many source files for this to be practical, we recommend use of the WHERE-IS Library module.

While the location of definitions is made known to the edit interface when files are loaded, it can be very inefficient when files are not SYSLOADed. If, for example, you load ten compiled files with LDFLG=NIL and then evaluate (ED 'FOO), then the COMS of all ten files must be searched for definitions of each manager type with name FOO. With forty manager types this comes to 400 parses of COMS -- a time-consuming operation. If you instead load the compiled files SYSLOAD and the sources PROP, then no COMS need be searched, as checking for definitions of each manager type is sufficient.

Loadflg argument

The Medley release contains a new keyword argument to **cl:load**.

(**cl:load** *filename* &key *verbose print if-does-not-exist loadflg*)

The *loadflg* argument follows the semantics of the loadflg argument to **il:load**, with the exception that the loadflg argument will always be interned in the Interlisp package.

EXAMPLE:

```
(cl:load "Mycompiled-file.dfasl" :loadflg :sysload)
```

In this example, "Mycompiled-file.dfasl" will load without the file manager noticing that file.

Note: As explained in the previous section, we discourage loading either sort of compiled file (lcom or dfasl) with any value for *ldflg* but SYSLOAD.

Changes in CL:MAP, CL:WRITE-STRING, CL:COERCE, CL:GENSYM and IL:DEFERREDCONSTANT

In Lyric, a compiled call to CL:MAP that had been used for effect would occasionally cons up a new list anyway. It would fail in the case that the first argument was a constant that evaluated to NIL, but not NIL itself, e.g. 'NIL. This has been fixed and no longer occurs in Medley.

CL:WRITE-STRING is now twice as fast and creates no new structure.

CL:COERCE now correctly returns the original object in all cases where Common Lisp and Lisp require it.

The CL Compiler now compiles CL:GENSYM properly.

IL:DEFERREDCONSTANT is now handled correctly by the XCL compiler.

ADD.PROCESS no longer coerces the process name to a symbol. Rather, process names are treated as case-insensitive strings. Thus, you can use strings for process names, and when typing process commands to an exec, you need not worry about getting the alphabetic case correct.

Compiler keeps Special &REST arguments

The CL Compiler now retains special &REST arguments. The Lyric compiler threw away special &REST arguments. This has been fixed in the Medley CL Compiler.

Compiler ignores TEdit formatting

COMPILE-FILE will now ignore TEdit formatting, but only if TEdit is loaded.

Compiler notices Tail-recursive Lexical Functions

The XCL Compiler now performs tail recursion elimination on FLETed lexical functions.

Compiler Error Message "BUG: Inconsistent stack depths seen"

You may occasionally see this error message while compiling. Normally, error messages from the compiler beginning with "BUG" indicate an internal compiler error. In this particular case, the compiler error may reflect an error in the code you are compiling.

There is currently no compile-time argument checking. The compiler performs an optimization that turns a tail-recursive function call into a jump back to the beginning of the function. If this tail-recursive call has the wrong number of arguments, the stack modeler in the assembler will detect this as inconsistent stack depths, leading to the above error message.

EXAMPLE:

```
(defun bad-length (x n)
  (if (endp x) n (bad-length (cdr x))))
```

Compiling this form will result in the error "BUG: Inconsistent stack depths seen." The recursive call to bad-length has only one argument, but the function expects two.

Thus, if you see this error message, you should check for tail-recursive function calls with the wrong number of arguments.

Format ~C and WRITE-CHAR

In accordance with a recommendation of X3J13, the ~C FORMAT operation with no modifiers now behaves exactly the same as WRITE-CHAR for characters with no bits. The Medley release of XCL conforms to this; the Lyric release did not. If you need to obtain the Lyric behavior of ~C, use ~:C.

WITH-OUTPUT-TO-STRING and WITH-INPUT-FROM-STRING

For consistency with WITH-OPEN-STREAM and WITH-OPEN-FILE, WITH-OUTPUT-TO-STRING and WITH-INPUT-FROM-STRING now close the stream on exit from the form. WITH-OUTPUT-TO-STRING is now significantly faster when writing long strings.

[This page intentionally left blank]

APPENDIX A. THE EXEC

In most Common Lisp implementations, there is a "top-level **read-eval-print** loop," which reads an expression, evaluates it, and prints the results. In Xerox Common Lisp, the Exec acts as the top-level loop, but in addition to **read-eval-print**, it also performs a number of other tasks, and allows a much greater range of inputs. This appendix contains information from the Lyric and Medley releases. Medley additions are indicated with revision bars in the right margin.

The Exec is based on concepts from the Interlisp Programmer's Assistant (see the *Interlisp-D Reference Manual*).

The Exec traps all throws, and recovers gracefully. It prints all values resulting from evaluation, on separate lines. When zero values are returned, nothing is printed.

The Exec keeps track of your previous input, in a structure called the history list. A history list is a list of the information associated with each of the individual events that have occurred, where each event corresponds to one input. Associated with each event on the history list is the input, its values, plus other optional information such as side-effects, formatting information, etc.

The following dialogue contains illustrative examples and gives the flavor of the use of the Exec. Be sure to type these examples to an Exec whose ***PACKAGE*** is set to the **XCL-USER** package. The Exec that Lisp starts up with is set to the **XCL-USER** package. Each prompt consists of an event number and a prompt character ("**>**").

```
12>(setq foo 5)
5
13>(setq foo 10)
10
14>undocr
SETQ undone.
15>foocr
5
```

*This is an example of direct communication with the Exec. You have instructed the Exec to **undo** the previous event.*

...

```
25>set(lst1 (a b c))
(A B C)
26>(setq lst2 '(d e f))
(D E F)
27>(mapc #'(lambda (x) (setf (get x 'myprop) t)) lst1)
(A B C)
```

*The Exec accepts input both in APPLY format (the **SET**) and EVAL format (the **SETQ**.) In event 27, the user adds a property **MYPROP** to the symbols **A**, **B**, and **C**.*

```
28>use lst2 for lst1 in 27cr
NIL
```

You just instructed the Exec to go back to event number 27, substitute **LST2** for **LST1**, and then re-execute the expression. You could have also used -2 instead of 27, specifying a relative address.

•
•
•

```
46>(setf my-hash-table (make-hash-table))
```

```
#<Hash-Table @ 66,114034>
```

```
47>(setf (gethash 'foo my-hash-table) (string 'foo))  
"FOO"
```

If **STRING** were computationally expensive (which it is not), then you might be caching its value for later use.

```
48>use fie for foo in stringcr  
"FIE"
```

You now decide you would like to redo the **SETF** with a different value. You specify the event using **"IN STRING"** rather than **SETF**.

```
49>?? usecr
```

```
      USE FIE FOR FOO IN STRING  
48> (SETF (GETHASH 'FIE MY-HASH-TABLE)  
      (STRING 'FIE))  
      "FIE"
```

Here you ask the Exec (using the **??** command) what it has on its history list for the last input. Since the event corresponds to a command, the Exec displays both the original command and the generated input.

The most common interaction with the Exec occurs at the top level or in the debugger, where you type in expressions for evaluation, and see the values printed out. In this mode, the Exec acts much like a standard Common Lisp top-level loop, except that before attempting to evaluate an input, the Exec first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or re-execution. The Exec also notes new functions and variables to be added to its spelling lists to enable future corrections.

After updating the history list, the Exec executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result. Finally the Exec displays a prompt to indicate it is again ready for input.

Input Formats

The Exec accepts three forms of input: an expression to be evaluated (EVAL-format), a function-name and arguments to apply it to (APPLY-format), and Exec commands, as follows:

EVAL-format input

If you type a single expression, either followed by a carriage-return, or, in the case of a list, terminated with balanced parenthesis, the expression is evaluated and the value is returned. For example, if the value of the variable **FOO** is the list **(A B C)**:

```
32>FOOcr  
(A B C)
```

Similarly, if you type a Lisp expression, beginning with a left parenthesis and terminated by a matching right parenthesis, the form is simply passed to **EVAL** for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; the reader will supply one automatically. If a carriage-return is typed before the final matching right parenthesis or bracket, it is treated the same as a space, and input continues. The following examples are interpreted identically:

```
123> (+ 1 (* 2 3))
7
```

```
124> (+ 1 (*cr
2 3))
7
```

APPLY-format input

Often, when typing at the keyboard, you call functions with constant argument values, which would have to be quoted if you typed them in "EVAL-format." For convenience, if you type a symbol immediately followed by a list form, the symbol is **APPLY**ed to the elements within the list, unevaluated. The input is terminated by the matching right parenthesis. For example, typing **LOAD(FOO)** is equivalent to typing **(LOAD 'FOO)**, and **GET(X COLOR)** is equivalent to **(GET 'X 'COLOR)**. As a simple special case, a single right parenthesis is treated as a balanced set of parentheses, e.g.

```
125>UNBREAK)
```

is equivalent to

```
125>UNBREAK()
```

The reader will only supply the "carriage return" automatically if no space appears between the initial symbol and the list that follows; if there is a space after the initial symbol on the line and the list that follows, the input is not terminated until a carriage return is explicitly typed.

Note that APPLY-format input cannot be used for macros or special forms.

Exec commands

The Exec recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list. The format of a command is always a line beginning with the command name. (The Exec looks up the command name independent of package, so that Exec commands are package independent.) The remainder of the line, if any, is treated as "arguments" to the command. For example,

```
128>UNDOcr
mapc undone
129>UNDO (FOO --)cr
foo undone
```

are all valid command inputs.

Multiple Execs and the Exec's Type

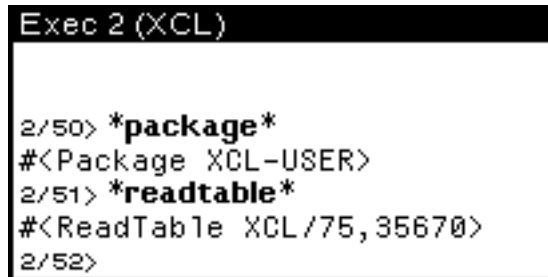
Multiple Execs

More than one Exec can be active at any one time. New Execs can be created by selecting the Exec menu item in the background pop-up menu. When a prompt is printed for an event in other than the first Exec, the prompt is preceded with the Exec number; for example:

2/50>

might be a prompt in Exec 2. All Execs share the same history list, but each event records which Exec it goes with. That is, although a single global list exists, the Xerox Lisp history system maintains the separate threads of control within each Exec.

Exec type Several variables are very important to an Exec since they control the format of reading and printing. Together these variables describe a type of exec. Put another way, this is the Exec's mode. To allow easier setting of these modes some standard bindings for the variables have been named. The names provide the user an Exec of the Common Lisp (CL), Interlisp (IL) or Xerox Extended Common Lisp (XCL) type. An Exec's type is usually displayed in the title bar of its window in parentheses:



```
Exec 2 (XCL)
2/50> *package*
#<Package XCL-USER>
2/51> *readtable*
#<ReadTable XCL/75,35670>
2/52>
```

Event Specification

Exec commands, like **UNDO**, frequently refer to previous events in the session's history. All Exec commands use the same conventions and syntax for indicating which event(s) the command refers to. This section shows you the syntax used to specify previous events.

An event address identifies one event on the history list. For example, the event address **42** refers to the event with event number 42, and **-2** refers to two events back in the *current* Exec. Usually, an event address will contain only one or two commands.

Event addresses can be concatenated. For example, if **FOO** refers to event *N*, **FOO FIE** will refer to the first event before event *N* which contains **FIE**.

The symbols used in event addresses (such as **AND**, **F**, **=**, etc. are compared with **STRING-EQUAL**, so that it does not matter what the current package is when you type an event address symbol to an Exec.

Event addresses are interpreted as follows:

- N* (an integer) If *N* is positive, it refers to the event with event number *N* (no matter which Exec the event occurred in.) If *N* is negative, it always refers to the event *-N* events backwards counting *only* events belonging to the *current* Exec.
- F** Specifies that the next object in the event address is to be searched for, regardless of what it is. For example, **F -2** looks for an event containing **-2**.
- =** Specifies that the next object is to be searched for in the *values* of events, instead of the inputs.

SUCHTHAT *PRED* Specifies an event for which the function *PRED* returns true. *PRED* should be a function of two arguments, the input portion of the event, and the event itself.

PAT Any other event address command specifies an event whose input contains an expression that matches *PAT*. When multiple Execs are active, all events are searched, no matter which Exec they belong to. The pattern can be a simple symbol, or a more complex search pattern.

Note: Specifications used below of the form *EventAddress_i* refer to event addresses, as described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in *EventAddress₁ AND EventAddress₂*, the notation *EventAddress₁* corresponds to all words up to the **AND** in the event specification, and *EventAddress₂* to all words after the **AND** in the event specification.

FROM *EventAddress* All events since *EventAddress*, inclusive. For example, if there is a single Exec and the current event is number 53, then **FROM 49** specifies events 49, 50, 51, and 52. **FROM** will include events from *all* Execs.

ALL *EventAddress* Specifies all events satisfying *EventAddress*. For example, **ALL LOAD, ALL SUCHTHAT FOO-P**.

empty If nothing is specified, it is the same as specifying **-1**, i.e., the last event in the current Exec.

EventSpec₁ AND EventSpec₂ AND . . . AND EventSpec_N

Each of the *EventSpec_i* is an event specification. The lists of events are concatenated. For example, **ALL MAPC AND ALL STRING AND 32** specifies all events containing **MAPC**, all containing **STRING**, and also event **32**. Duplicate events are removed.

Exec Commands

All Exec commands are input as lines which begin with the name of the command. The name of an Exec command is not a symbol and therefore is not sensitive to the setting of the current package (the value of ***PACKAGE***).

EventSpec is used to denote an event specification which in most cases will be either a specific event address (e.g., 42) or a relative one (e.g., -3). Unless specified otherwise, omitting *EventSpec* is the same as specifying *EventSpec=-1*. For example, **REDO** and **REDO -1** are the same.

REDO *EventSpec*

[Exec command]

Redoes the event or events specified by *EventSpec*. For example, **REDO 123** redoes the event numbered 123.

RETRY *EventSpec*

[Exec command]

Similar to **REDO** except sets the debugger parameters so that any errors that occur while executing *EventSpec* will cause breaks.

USE *NEW* [FOR *OLD*] [IN *EventSpec*]

[Exec command]

Substitutes *NEW* for *OLD* in the events specified by *EventSpec*, and redoes the result. *NEW* and *OLD* can include lists or symbols, etc.

For example, **USE SIN (- X) FOR COS X IN -2 AND -1** will substitute **SIN** for every occurrence of **COS** in the previous two events, and substitute **(- X)** for every occurrence of **X**, and reexecute them. (The substitutions do not change the previous information saved about these events on the history list.)

If **IN *EventSpec*** is omitted, the first member of *OLD* is used to search for the appropriate event. For example, **USE DEFAULTFONT FOR DEFLATFONT** is equivalent to **USE DEFAULTFONT FOR DEFLATFONT IN F DEFLATFONT**. The **F** is inserted to handle correctly the case where the first member of *OLD* could be interpreted as an event address command.

If *OLD* is omitted, substitution is for the "operator" in that command. For example **FBOUNDP(FF)** followed by **USE CALLS** is equivalent to **USE CALLS FOR FBOUNDP IN -1**.

If *OLD* is not found, **USE** will print a question mark, several spaces and the pattern that was not found. For example, if you specified **USE Y FOR X IN 104** and *X* was not found, "X ?" is printed to the Exec.

You can also specify more than one substitution simultaneously as follows:

USE *NEW*₁ FOR *OLD*₁ AND ... AND *NEW*_{*N*} FOR *OLD*_{*N*} [IN *EventSpec*]

[Exec command]

Note: The **USE** command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, **USE FOR FOR AND AND AND FOR FOR** will be parsed correctly.

Every **USE** command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification that defines the input expression in which the substitution takes place. If the **USE** command has the same number of expressions as arguments, the substitution procedure is straightforward. For example, **USE X Y FOR U V** means substitute **X** for **U** and **Y** for **V**, and is equivalent to **USE X FOR U AND Y FOR V**.

However, the **USE** command also permits distributive substitutions for substituting several expressions for the same argument. For example, **USE A B C FOR X** means first substitute **A** for **X** then substitute **B** for **X** (in a new copy of the expression), then substitute **C** for **X**. The effect is the same as three separate **USE** commands.

Similarly, **USE A B C FOR D AND X Y Z FOR W** is equivalent to **USE A FOR D AND X FOR W**, followed by **USE B FOR D AND Y**

FOR W, followed by **USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y** also corresponds to three substitutions, the first with **A** for **D** and **X** for **Y**, the second with **B** for **D**, and **X** for **Y**, and the third with **C** for **D**, and again **X** for **Y**. However, **USE A B C FOR D AND X Y FOR Z** is ambiguous and will cause an error.

Essentially, the **USE** command operates by proceeding from left to right handling each **AND** separately. Whenever the number of expressions exceeds the number of expressions available, multiple **USE** expressions are generated. Thus **USE A B C D FOR E F** means substitute **A** for **E** at the same time as substituting **B** for **F**, then in another copy of the indicated expression, substitute **C** for **E** and **D** for **F**. This is also equivalent to **USE A C FOR E AND B D FOR F**.

Note: The **USE** command correctly handles the situation where one of the old expressions is the same as one of the new ones, **USE X Y FOR Y X**, or **USE X FOR Y AND Y FOR X**.

? &OPTIONAL NAME [Exec command]

If *NAME* is not provided describes all available Exec commands by printing the name, argument list, and description of each. With *NAME*, only that command is described.

?? EventSpec [Exec command]

Prints the most recent event matching the given *EventSpec*.

CONN DIRECTORY [Exec command]

Changes default pathname to *DIRECTORY*.

DA [Exec command]

Returns current date and time.

DIR &OPTIONAL PATHNAME &REST KEYWORDS [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

DO-EVENTS &REST INPUTS &ENVIRONMENT ENV [Exec command]

DO-EVENTS is intended as a way of putting together several different events, which can include commands. It executes the multiple *INPUTS* as a single event. The values returned by the **DO-EVENTS** event are the concatenation of the values of the inputs. An input is not an *EventSpec*, but a call to a function or command. If *ENV* is provided it is a lexical environment in which all evaluations (functions and commands) will take place. Event specification in the *INPUTS* should be explicit, not relative, since referring to the last event will reinvoked the executing **DO-EVENTS** command.

FIX &REST <i>EventSpec</i>	[Exec command]
Edits the specified event prior to reexecuting it. If the number of characters in the Fixed line is less than the variable TTYINFIXLIMIT then it will be edited using TTYIN, otherwise the Lisp editor is called via EDITE .	
FORGET &REST <i>EventSpec</i>	[Exec command]
Erases UNDO information for the specified events.	
NAME <i>COMMAND-NAME</i> &OPTIONAL <i>ARGUMENTS</i> &REST <i>EVENT-SPEC</i>	[Exec command]
Defines a new command, <i>COMMAND-NAME</i> , and its <i>ARGUMENTS</i> , containing the events in <i>EVENT-SPEC</i> .	
NDIR &OPTIONAL <i>PATHNAME</i> &REST <i>KEYWORDS</i>	[Exec command]
Shows a directory listing for <i>PATHNAME</i> or the connected directory in abbreviated format. If provided, <i>KEYWORDS</i> indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.	
PL <i>SYMBOL</i>	[Exec command]
Prints the property list of <i>SYMBOL</i> in an easy to read format.	
REMEMBER &REST <i>EVENT-SPEC</i>	[Exec command]
Tells File Manager to remember type-in from specified event(s) , <i>EVENT-SPEC</i> , as expressions to save.	
SHH &REST <i>LINE</i>	[Exec command]
Executes <i>LINE</i> without history list processing.	
UNDO &REST <i>EventSpec</i>	[Exec command]
Undoes the side effects of the specified event (see below under "Undoing").	
PP &OPTIONAL <i>NAME</i> &REST <i>TYPES</i>	[Exec command]
Shows (prettyprinted) the definitions for <i>NAME</i> specified by <i>TYPES</i> .	
SEE &REST <i>FILES</i>	[Exec command]
Prints the contents of <i>FILES</i> in the Exec window, hiding comments.	
SEE* &REST <i>FILES</i>	[Exec command]
Prints the contents of <i>FILES</i> in the Exec window, showing comments.	

TIME FORM & KEY REPEAT & ENVIRONMENT ENV [Exec command]

Times the evaluation of *FORM* in the lexical environment *ENV*, repeating *REPEAT* number of times. Information is displayed in the Exec window.

TY & REST FILES [Exec command]

Exactly like the **TYPE** Exec command.

TYPE & REST FILES [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

Variables

A number of variables are provided for convenience in the Exec.

IL:IT [Variable]

Whenever an event is completed, the global value of the variable **IT** is reset to the event's value. For example,

```
312>(SQRT 2)
1.414214
313>(SQRT IL:IT)
1.189207
```

Following a **??** command, **IL:IT** is set to the value of the last event printed. The inspector has an option for setting the variable **IL:IT** to the current selection or inspected object, as well. The variable **IL:IT** is global, and is shared among all Execs. **IL:IT** is a convenient mechanism for passing values from one process to another.

Note: **IT** is in the INTERLISP package and these examples are intended for an Exec whose ***PACKAGE*** is set to **XCL-USER**. Thus, **IT** must be package qualified (the **IL:**).

The following variables are maintained independently by each Exec. (When a new Exec is started, the initial values are **NIL**, or, for a nested Exec, the value for the "parent" Exec. However, events executed under a nested Exec will not affect the parent values.)

CL:- [Variable]

CL:+ [Variable]

CL:++ [Variable]

CL:+++ [Variable]

While a form is being evaluated by the Exec, the variable **-** is bound to the form, **CL:+** is bound to the previous form, **CL:++** the one before, etc. If the input is in apply-format rather than eval-format, the value of the respective variable is just the function name.

CL:* [Variable]

CL:** [Variable]

CL:*** [Variable]

While a form is being evaluated by the Exec, the variable **CL:*** is bound to the (first) value returned by the last event, **CL:**** to the event before that, etc. The variable **CL:*** differs from **IT** in that **IT** is global while each separate Exec maintains its own copy of **CL:***, **CL:**** and **CL:*****. In addition, the history commands change **IT**, but only inputs which are retained on the history list can change **CL:***.

CL:/ [Variable]

CL:// [Variable]

CL:/// [Variable]

While a form is being evaluated by an Exec, the variable **CL:/** is bound to a list of the results of the last event in that Exec, **CL://** to the values of the event before that, etc.

Fonts in the Exec

The Exec can use different fonts for displaying the prompt, user's input, intermediate printout, and the values returned by evaluation. The following variables control the Exec's font use:

PROMPTFONT [Variable]

Font used for printing the event prompt.

INPUTFONT [Variable]

Font used for echoing user's type-in.

PRINTOUTFONT [Variable]

Font used for any intermediate printing caused by execution of a command or evaluation of a form. Initially the same as **DEFAULTFONT**.

VALUEFONT [Variable]

Font used to print the values returned by evaluation of a form. Initially the same as **DEFAULTFONT**.

Changing the Exec

(CHANGESLICE *N HISTORY* —) [Function]

Changes the time-slice of the history list *HISTORY* to *N*. If **NIL**, *HISTORY* defaults to the top level history **LISPXHISTORY**.

Note: The effect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, **CHANGESLICE** is undoable, so that these events are (temporarily) recoverable. Therefore, if you want to recover the storage associated with these events without waiting *N* more events until the **CHANGESLICE** event drops off the history list, you must perform a **FORGET** command.

Defining New Commands

You can define new Exec commands using the **XCL:DEFCOMMAND** macro.

(XCL:DEFCOMMAND NAME ARGUMENT-LIST &REST BODY) [Macro]

XCL:DEFCOMMAND is similar to **XCL:DEFMACRO**, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, defstructure, and use all of the features of macro argument lists. When *NAME* is subsequently typed to the Exec, the rest of the line is processed like the arguments to a macro, and the *BODY* is executed. **XCL:DEFCOMMAND** is a definer; the File Manager will remember typed-in definitions and allow them to be saved, edited with **EDITDEF**, etc.

There are actually three kinds of commands that can be defined, **:EVAL**, **:QUIET**, and **:INPUT**. Commands can also be marked as only for the debugger, in which case they are labelled as **:DEBUGGER**. The command type is noted by supplying a list for the *NAME* argument to **XCL:DEFCOMMAND**, where the first element of the list is the command name, and the other elements are keyword(s) for the command type and, optionally **:DEBUGGER**.

Note: The documentation string in user defined Exec commands is automatically added to the documentation descriptions by the **CL:DOCUMENTATION** function under the **COMMANDS** type and can be shown using the **? Exec** command.

:EVAL This is the default. The body of the command just gets executed, and its value is the value of the event. For example (in an XCL Exec),

```
(DEFCOMMAND (LS :EVAL)
(&OPTIONAL (NAMESTRING *DEFAULT-PATHNAME-DEFAULTS*)
&REST DIRECTORY-KEYWORDS)
(MAPC
  #'(LAMBDA (PATHNAME) (FORMAT T "~&~A" (NAMESTRING PATHNAME)))
  (APPLY #'DIRECTOR NAMESTRING DIRECTORY-KEYWORDS))
(VALUES))
```

would define the **LS** command to print out all file names that match the input namestring. The **(VALUES)** means that no value will be printed by the event, only the intermediate output from the **FORMAT**.

:QUIET These commands are evaluated, but neither your input nor the results of the command are stored on the history list. For example, the **??** and **SHH** commands are quiet.

:INPUT These commands work more like macros, in that the result of evaluating the command is treated as a new line of input. The **FIX** command is an input command. The result is treated as a line; a single expression in EVAL-format should be returned as a list of the expression to **EVAL**.

The new Exec now will not consider unparenthesized input with more than one argument to be in apply format. This is the same behavior as the older execs, e.g.:

list(1) ; is apply format (executes after close paren is typed)

list (1) ; is apply format (second arg is a list, no trailing args given)

list '(1) 2 3 ; is NOT apply format, arguments are evaluated

list 1 2 3 ; is NOT apply format, arguments are evaluated

list 1 ; not legal input: second argument is not a list

Undoing

Note: This discussion only applies to undoing under the Exec, Debugger and within the **UNDOABLY** macro; editors handle undoing in a different fashion.

The **UNDO** facility allows recording of destructive changes such that they can be played back to restore a previous state. There are two kinds of **UNDO**ing: one is done by the Exec, the other is available for use in a programmer's code. Both methods share information about what kind of operations can be undone and where the changes are recorded.

Undoing in the Exec

UNDO *EventSpec*

[Exec command]

The Exec's **UNDO** command is implemented by watching the evaluation of forms and requiring undoable operations in that evaluation to save enough information on the history list to reverse their side effects. The Exec simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The **UNDO** command works on itself the same way: it recovers the saved information and performs the corresponding inverses. Thus, **UNDO** is effective on itself, so that you can **UNDO** an **UNDO**, and **UNDO** that, etc.

Only when you attempt to undo an operation does the Exec check to see whether any information has been saved. If none has been saved, and you have specifically named the event you want undone, the Exec types **nothing saved**. (When you just type **UNDO**, the Exec only tries to undo the last operation.)

UNDO watches evaluation using **CL:EVALHOOK** (thus, calling **CL:EVALHOOK** cannot be undone). Each form given to **EVAL** is

examined against the list **LISPFNS** to see if it has a corresponding undoable version. If an undoable version of a call is found, it is called with the same arguments instead of the original. Therefore, before evaluating all subforms of your input, the Exec substitutes the corresponding undoable call for any destructive operation. For example, if you type **(DEFUN FOO ...)**, undoable versions of the forms that set the definition into the symbol function cell are evaluated. **FOO**'s function definition itself is not made undoable.

Undoing in Programs

There are two ways to make a program undoable. The simplest method is to wrap the program's form in the **UNDOABLY** macro. The other is to call undoable versions of destructive operations directly.

(XCL:UNDOABLY &REST FORMS) [Macro]

Executes the forms in *FORMS* using undoable versions of all destructive operations. This is done by "walking" (see **WALKFORM**) all of the *FORMS* and rewriting them to use the undoable versions of destructive operations (**LISPFNS** makes the association).

(STOP-UNDOABLY &REST FORMS) [Macro]

Normally executes as **PROGN**; however, within an **UNDOABLY** form, explicitly causes *FORMS* not to be done undoably. Turns off rewriting of the *FORMS* to be undoable inside an **UNDOABLY** macro.

Undoable Versions of Common Functions

Efficiency and overhead are serious considerations for the execution of a user program. Thus, the programmer may need more control over the saving of undo information than that provided by the **UNDOABLY** macro.

To make a function undoable, you can simply substitute the corresponding undoable function if you want to make a destructive operation in your own program undoable. When the undoable function is called, it will save the undo information in the current event on the history list.

Various operations, most notably **SETF**, have undoable versions. The following undoable macros are initially available:

UNDOABLY-POP

UNDOABLY-PUSH

UNDOABLY-PUSHNEW

UNDOABLY-REMF

UNDOABLY-ROTATEF

UNDOABLY-SHIFTF

UNDOABLY-DECF

UNDOABLY-INCF

UNDOABLY-SET-SYMBOL
UNDOABLY-MAKUNBOUND
UNDOABLY-FMAKUNBOUND
UNDOABLY-SETQ
XCL:UNDOABLY-SETF
UNDOABLY-PSETF
UNDOABLY-SETF-SYMBOL-FUNCTION
UNDOABLY-SETF-MACRO-FUNCTION

Note: Many destructive Common Lisp functions do not currently have undoable versions, e.g., **CL:NREVERSE**, **CL:SORT**, etc. The current list of undoable functions is saved on the association list **LISPXFNS**.

Modifying the UNDO Facility

You will usually wish to extend the **UNDO** facility after creating a form whose side effects it might be desirable to undo, for instance a file renaming function.

An undoable version of the function needs to be written. This can be done by explicitly saving previous state information away, or by renaming calls in the function to their undoable equivalent. Undo information should be saved on the history list using **IL:UNDOSAVE**.

You must then hook the undoable version of the function into the undo facility. You do this by either using the **IL:LISPXFNS** association list, or in the case of a **SETF** modifier, on the **IL:UNDOABLE-SETF-INVERSE** property of the **SETF** function.

LISPXFNS

[Variable]

Contains an association list which maps from destructive operations to their undoable form. Initially this list contains:

((CL:POP . UNDOABLY-POP)
(CL:PSETF . UNDOABLY-PSETF)
(CL:PUSH . UNDOABLY-PUSH)
(CL:PUSHNEW . UNDOABLY-PUSHNEW)
((CL:REMF) . UNDOABLY-REMF)
(CL:ROTATEF . UNDOABLY-ROTATEF)
(CL:SHIFTF . UNDOABLY-SHIFTF)
(CL:DECF . UNDOABLY-DECF)
(CL:INCF . UNDOABLY-INCF)
(CL:SET . UNDOABLY-SET-SYMBOL)
(CL:MAKUNBOUND . UNDOABLY-MAKUNBOUND)
(CL:FMAKUNBOUND . UNDOABLY-FMAKUNBOUND)
. . . plus the original Interlisp undo associations)

(XCL:UNDOABLY-SETF PLACE VALUE ...)

[Macro]

Like **CL:SETF** but saves information so it may be undone. **UNDOABLY-SETF** uses undoable versions of the setf function located on the **UNDOABLE-SETF-INVERSE** property of the function being **SETF**ed. Initially these **SETF** names have such a property:

CL:SYMBOL-FUNCTION - **UNDOABLY-SETF-SYMBOL-FUNCTION**

CL:MACRO-FUNCTION - **UNDOABLY-SETF-MACRO-FUNCTION**

(UNDOABLY-SETQ &REST FORMS)

[Function]

Typed-in **SETQ**s (and **SETF**s on symbols) are made undoable by substituting a call to **UNDOABLY-SETQ**. **UNDOABLY-SETQ** operates like **SETQ** on lexical variables or those with dynamic bindings; it only saves information on the history list for changes to global, "top-level" values.

(UNDOSAVE UNDOFORM HISTENTRY)

[Function]

Adds the undo information *UNDOFORM* to the **SIDE** property of the history event *HISTENTRY*. If there is no **SIDE** property, one is created. If the value of the **SIDE** property is **NOSAVE**, the information is not saved. *HISTENTRY* specifies an event. If *HISTENTRY*=**NIL**, the value of **LISPXHIST** is used. If both *HISTENTRY* and **LISPXHIST** are **NIL**, **UNDOSAVE** is a no-op.

The form of *UNDOFORM* is *(FN . ARGS)*. Undoing is done by performing **(APPLY (CAR UNDOFORM) (CDR UNDOFORM))**.

#UNDOSAVES

[Variable]

The value of **#UNDOSAVES** is the maximum number of *UNDOFORM*s to be saved for a single event. When the count of *UNDOFORM*s reaches this number, **UNDOSAVE** prints the message **CONTINUE SAVING?**, asking if you want to continue saving. If you answer **NO** or default, **UNDOSAVE** discards the previously saved information for this event, and makes **NOSAVE** be the value of the property **SIDE**, which disables any further saving for this event. If you answer **YES**, **UNDOSAVE** changes the count to -1, which is then never incremented, and continues saving. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone.

If **#UNDOSAVES** is negative, then when the count reaches **(ABS #UNDOSAVES)**, **UNDOSAVE** simply stops saving without printing any messages or other interactions. **#UNDOSAVES**=**NIL** is equivalent to **#UNDOSAVES**=infinity. **#UNDOSAVES** is initially **NIL**.

The configuration described here has been found to be a very satisfactory one. You pay a very small price for the ability to undo what you type in, since the interpreted evaluation is simply watched for destructive operations, or if you wish to protect yourself from malfunctioning in your own programs, you can explicitly call, or have your program rewritten to explicitly call, undoable functions.

Undoing Out of Order

UNDOABLY-SETF operates undoably by saving (on the history list) the cell that is to be changed and its original contents. Undoing an **UNDOABLY-SETF** restores the saved contents.

This implementation can produce unexpected results when multiple modifications are made to the same piece of storage and then undone out of order. For example, if you type **(SETF (CAR FOO) 1)**, followed by **(SETF (CAR FOO) 2)**, then undo both events by undoing the most recent event first, then undoing the older event, **FOO** will be restored to its state before either event operated. However if you undo the first event, *then* the second event, **(CAR FOO)** will be **1**, since this is what was in **CAR** of **FOO** before **(UNDOABLY-SETF (CAR FOO) 2)** was executed. Similarly, if you type **(NCONC FOO '(1))**, followed by **(NCONC FOO '(2))**, undoing just **(NCONC FOO '(1))** will remove both **1** and **2** from **FOO**. The problem in both cases is that the two operations are not independent.

In general, operations are always independent if they affect different lists or different sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the right thing. However, undoing dependent operations out of order may not always have the predicted effect.

Format and Use of the History List

LISPXHISTORY

[Variable]

The Exec currently uses one primary history list, **LISPXHISTORY** for the storing events.

The history list is in the form **(EVENTS EVENT# SIZE MOD)**, where **EVENTS** is a list of events with the most recent event first, **EVENT#** is the event number for the most recent event on **EVENTS**, **SIZE** is the the maximum length **EVENTS** is allowed to grow. **MOD** is is the maximum event number to use, after which event numbers roll over. **LISPXHISTORY** is initialized to **(NIL 0 100 1000)**.

The history list has a maximum length, called its time-slice. As new events occur, existing events are aged, and the oldest events are forgotten. The time-slice can be changed with the function **CHANGESLICE**. Larger time-slices enable longer memory spans, but tie up correspondingly greater amounts of storage. Since a user seldom needs really ancient history, a relatively small time-slice such as 30 events is usually adequate, although some users prefer to set the time-slice as large as 200 events.

Each individual event on **EVENTS** is a list of the form **(INPUT ID VALUE . PROPS)**. For Exec events, **ID** is a list **(EVENT-NUMBER EXEC-ID)**. The **EVENT-NUMBER** is the number of the event, while the **EXEC-ID** is a string that uniquely identifies the Exec. (The **EXEC-ID** is used to identify which events belong to the "same" Exec.) **VALUE** is the (first) value of the event. **PROPS** is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that the user typed-in. For an APPLY-format input this is a list consisting of two expressions; for an EVAL-format input, this is a list of just one expression; for an input entered as list of atoms, *INPUT* is simply that list. For example,

User Input	<i>INPUT</i> is:
LIST(1 2)	(LIST (1 2))
(LIST 1 1)	((LIST 1 1))

DIR "{DSK}<LISPFILES>"^{cr}	(DIR "{DSK}<LISPFILES>")
--	---------------------------------------

If you type in an Exec command that executes other events (**REDO**, **USE**, etc.), several events might result. When there is more than one input, they are wrapped together into one invocation of the **DO-EVENTS** command.

The same convention is used for representing multiple inputs when a **USE** command involves sequential substitutions. For example, if you type **FBOUNDP(FOO)** and then **USE FIE FUM FOR FOO**, the input sequence that will be constructed is **DO-EVENTS (EVENT FBOUNDP (FIE)) (EVENT FBOUNDP (FUM))**, which is the result of substituting **FIE** for **FOO** in **(FBOUNDP (FOO))** concatenated with the result of substituting **FUM** for **FOO** in **(FBOUNDP (FOO))**.

PROPS is a property list of the form *(PROPERTY₁ VALUE₁ PROPERTY₂ VALUE₂ ...)*, that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the Exec:

SIDE	A list of the side effects of the event. See UNDOSAVE .
LISPXPRINT	Used to record calls to EXEC-FORMAT , and printed by the ?? command.

Making or Changing an Exec

(XCL:ADD-EXEC &KEY PROFILE REGION TTY ID)	[Function]
--	------------

Creates a new process and window with an Exec running in it. *PROFILE* is the type of the Exec to be created (see below under XCL:SET-EXEC-TYPE). *REGION* optionally gives the shape and location of the window to be used. If not provided the user will be prompted. *TTY* is a flag, which, if true, causes the tty to be given to the new Exec process. *ID* is a string identifier to use for events generated in this exec. *ID* defaults to the number given to the Exec process created.

(XCL:EXEC &KEY WINDOW PROMPT COMMAND-TABLES ENVIRONMENT PROFILE TOP-LEVEL-P TITLE FUNCTION ID)	[Function]
---	------------

This is the main entry to the Exec. The arguments are:

WINDOW defaults to the current TTY display stream, or can be provided a window in which the Exec will run.

PROMPT is the prompt to print.

COMMAND-TABLES is a list of hash-tables for looking up commands (e.g., ***EXEC-COMMAND-TABLE*** or ***DEBUGGER-COMMAND-TABLE***).

ENVIRONMENT is a lexical environment used to evaluate things in.

READTABLE is the default readtable to use (defaults to the "Common Lisp" readtable).

PROFILE is a way to set the Exec's type (see above, "Multiple Execs and the Exec's Type").

TOP-LEVEL-P is a boolean, which should be true if this Exec is at the top level.

TITLE is an identifying title for the window title of the Exec.

FUNCTION is a function used to actually evaluate events, default is **EVAL-INPUT**.

ID is a string identifier to use for events generated in this Exec. *ID* defaults to the number given to the Exec process.

XCL:*PER-EXEC-VARIABLES*

[Variable]

A list of pairs of the form (*VAR INIT*). Each time an Exec is entered, the variables in ***PER-EXEC-VARIABLES*** are rebound to the value returned by evaluating *INIT*. The initial value of ***PER-EXEC-VARIABLES*** is:

```
(( (*PACKAGE* *PACKAGE*)
  (* *)
  (** **)
  (***) (***)
  (+ +)
  (++) ++
  (+++ +++)
  (- -)
  (/ /)
  (// //)
  (/// ///)
  (HELPFLAG T)
  (*EVALHOOK* NIL)
  (*APPLYHOOK* nil)
  (*ERROR-OUTPUT* *TERMINAL-IO*)
  (*READTABLE* *READTABLE*)
  (*package* *package*)
  (*eval-function* *eval-function*)
  (*exec-prompt* *exec-prompt*)
  (*debugger-prompt* *debugger-prompt*))
```

Most of these cause the values to be (re)bound to their current value in any inferior Exec, or to **NIL**, their value at the "top level".

XCL:*EVAL-FUNCTION*

[Variable]

Bound to the function used by the Exec to evaluate input. Typically in an INTERLISP Exec this is **IL:EVAL**, and in a Common Lisp Exec, **CL:EVAL**.

XCL:*EXEC-PROMPT*

[Variable]

Bound to the string printed by the Exec as a prompt for input. Typically in an INTERLISP Exec this is " ← ", and in a Common Lisp Exec, "> ".

XCL:*DEBUGGER-PROMPT*

[Variable]

Bound to the string printed by the debugger Exec as a prompt for input. Typically in an INTERLISP Exec this is " ← : ", and in a Common Lisp Exec, ": ".

(XCL:EXEC-EVAL FORM &OPTIONAL ENVIRONMENT)

[Function]

Evaluates *FORM* (using **EVAL**) in the lexical environment *ENVIRONMENT* the same as though it were typed in to **EXEC**, i.e., the event is recorded, and the evaluation is made undoable by substituting the UNDOABLE-functions for the corresponding destructive functions. **XCL:EXEC-EVAL** returns the value(s) of the form, but does not print it, and does not reset the variables *, **, ***, etc.

(XCL:EXEC-FORMAT CONTROL-STRING &REST ARGUMENTS)

[Function]

In addition to saving inputs and values, the Exec saves many system messages on the history list. For example, **FILE CREATED** ..., **FN redefined**, **VAR reset**, output of **TIME**, **BREAKDOWN**, **ROOM**, save their output on the history list, so that when ?? prints the event, the output is also printed. The function **XCL:EXEC-FORMAT** can be used in user code similarly. **XCL:EXEC-FORMAT** performs (APPLY #'CL:FORMAT *TERMINAL-IO* *CONTROL-STRING ARGUMENTS*) and also saves the format string and arguments on the history list associated with the current event.

(XCL:SET-EXEC-TYPE NAME)

[Function]

Sets the type of the current Exec to that indicated by *NAME*. This can be used to set up the Exec to your liking. *NAME* may be an atom or string. Possible names are:

INTERLISP, IL

READTABLE INTERLISP

PACKAGE INTERLISP

XCL:*DEBUGGER-PROMPT* "←: "

XCL:*EXEC-PROMPT* "←"

XCL:*EVAL-FUNCTION* IL:EVAL

XEROX-COMMON-LISP, XCL

READTABLE XCL

PACKAGE XCL-USER

XCL:*DEBUGGER-PROMPT* ": "

XCL:*EXEC-PROMPT* "> "

XCL:*EVAL-FUNCTION* CL:EVAL

COMMON-LISP, CL	*READTABLE* LISP *PACKAGE* USER XCL: *DEBUGGER-PROMPT* ": " XCL: *EXEC-PROMPT* "> " XCL: *EVAL-FUNCTION* CL:EVAL
OLD-INTERLISP-T	*READTABLE* OLD-INTERLISP-T *PACKAGE* INTERLISP XCL: *DEBUGGER-PROMPT* "←: " XCL: *EXEC-PROMPT* ": " XCL: *EVAL-FUNCTION* IL:EVAL

(XCL:SET-DEFAULT-EXEC-TYPE NAME)

[Function]

Like **XCL:SET-EXEC-TYPE** , but sets the type of Execs created by default, as from the background menu. Initially **XCL**. This can be used in your greet file to set default Execs to your liking.

Editing Exec Input

The Exec features an editor for input which provides completion, spelling correction, help facility, and character-level editing. The implementation is borrowed from the Interlisp module **TTYIN**. This section describes the use of the **TTYIN** editor from the perspective of the Exec.

Editing Your Input

Some editing operations can be performed using any of several characters; characters that are interrupts will, of course, not be read, so several alternatives are given. The following characters may be used to edit your input:

CONTROL-A, BACKSPACE	Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.
CONTROL-W	Deletes a "word". Generally this means back to the last space or parenthesis.
CONTROL-Q	Deletes the current line, or if the current line is blank, deletes the previous line.
CONTROL-R	Refreshes the current line. Two in a row refreshes the whole buffer (when doing multiline input).
ESCAPE	Tries to complete the current word from the spelling list USERWORDS . In the case of ambiguity, completes as far as is uniquely determined, or beeps.

UNDO key (on 1108 and 1186) Middle-blank key (on 1132)	Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed; when typed in the middle of a line fills in the remaining text from the old line; when typed following CONTROL-Q or CONTROL-W restores what those commands erased.
CONTROL-X	<p>Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced.</p> <p>If you are already at the end of the input and the expression is balanced except for lacking one or more right parentheses, CONTROL-X adds the required right parentheses to balance and returns.</p> <p>During most kinds of input, lines are broken, if possible, so that no word straddles the end of the line. The pseudo-carriage return ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You will not get carriage returns in your strings unless you explicitly type them.</p>

Using the Mouse

	Editing with the mouse during TTYIN input is slightly different than with other modules. The mouse buttons are interpreted as follows during TTYIN input:
<i>LEFT</i>	Moves the caret to where the cursor is pointing. As you hold down <i>LEFT</i> , the caret moves around with the cursor; after you let up, any type-in will be inserted at the new position.
<i>MIDDLE</i> or <i>LEFT+RIGHT</i>	Like <i>LEFT</i> , but moves only to word boundaries.
<i>RIGHT</i>	<p><i>Deletes</i> text from the caret to the cursor, either forward or backward. While you hold down <i>RIGHT</i>, the text to be deleted is inverted; when you let up, the text goes away. If you let up outside the scope of the text, nothing is deleted (this is how to cancel this operation).</p> <p>If you hold down <i>MOVE</i>, <i>COPY</i>, <i>SHIFT</i> or <i>CTRL</i> while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. The selection is made by holding the appropriate key down while pressing the mouse buttons <i>LEFT</i> (to select a character) or <i>MIDDLE</i> (to select a word), and optionally extend the selection either left or right using <i>RIGHT</i>. While you are doing this, the caret does not move, but the selected text is highlighted in a manner indicating what is about to happen. When the selection is complete, release the mouse buttons and then lift up on <i>MOVE/COPY/CTRL/SHIFT</i> and the appropriate action will occur:</p>
<i>COPY</i> or <i>SHIFT</i>	The selected text is inserted as if it were typed. The text is highlighted with a broken underline during selection.
<i>CTRL</i>	The selected text is deleted. The text is complemented during selection.
<i>MOVE</i> or <i>CTRL+SHIFT</i>	Combines copy and delete. The selected text is moved to the caret.

You can cancel a selection in progress by pressing *LEFT* or *MIDDLE* as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing the UNDO key (on the Xerox 1108/1186/1185) or the Middle-blank key (on the Xerox 1132). This is the same key that retrieves the previous buffer when issued at the end of a line.

Editing Commands

A number of characters have special effects while typing to the Exec. Some of them merely move the caret inside the input stream. While caret positioning can often be done more conveniently with the mouse, some of the commands, such as the case changing commands, can be useful for modifying the input.

In the descriptions below, current word means the word the cursor is under, or if under a space, the previous word. Currently, parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. The notation *[CHAR]* means meta-*CHAR*. The notation \$ stands for the ESCAPE/EXPAND key. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the meta key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the argument. Most of these commands are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands

[bs]	Backs up one (or n) characters.
[space]	Moves forward one (or n) characters.
[^]	Moves up one (or n) lines.
[f]	Moves down one (or n) lines.
[f]	Moves back one (or n) words.
[]]	Moves ahead one (or n) words.
[tab]	Moves to end of line; with an argument moves to nth end of line; [\$tab] goes to end of buffer.
[control-L]	Moves to start of line (or nth previous, or start of buffer).
[{] and []]	Goes to start and end of buffer, respectively (like [\$control-L] and [\$tab]).
[[] (meta-left-bracket)	Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Assorted Flags".)
[]] (meta-right-bracket)	Moves to end of current list.
[Sx]	Skips ahead to next (or nth) occurrence of character x, or rings the bell.
[Bx]	Backward search, i.e., short for [-S] or [-nS].

Buffer Modification Commands

- [Zx] Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command.
- [A] or [R] Repeats the last S, B, or Z command, regardless of any intervening input.
- [K] Kills the character under the cursor, or n chars starting at the cursor.
- [cr] When the buffer is empty is the same as undo i.e. restores buffer's previous contents. Otherwise is just like a <cr> (except that it also terminates an insert). Thus, [<cr><cr>] will repeat the previous input (as will undo<cr> without the meta key).
- [O] Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.
- [T] Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle odd cases, such as tabs.
- [G] Grabs the contents of the previous line from the cursor position onward. [nG] grabs the nth previous line.
- [L] Puts the current word, or n words on line, in lower case. [\$L] puts the rest of the line in lower case; or if given at the end of line puts the entire line in lower case.
- [U] Analogous to [L], for putting word, line, or portion of line in upper case.
- [C] Capitalizes. If you give it an argument, only the first word is capitalized; the rest are just lowercased.
- [control-Q] Deletes the current line. [\$control-Q] deletes from the current cursor position to the end of the buffer. No other arguments are handled.
- [control-W] Deletes the current word, or the previous word if sitting on a space.

Miscellaneous Commands

- [P] Prettyprints buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.
- [N] Refreshes line. Same as control-R. [\$N] refreshes the whole buffer; [nN] refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the window; in some circumstances, you may need to refresh the line for best results.
- [control-Y] Gets an Interlisp Exec.
- [\$control-Y] Gets an Interlisp Exec, but first unread the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for Interlisp, you can do [control-L\$control-Y] and give it to Lisp.
- [←] Adds the current word to the spelling list **USERWORDS**. With zero argument, removes word. See **TTYINCOMPLETEFLG**.

Useful Macros

If the event is considered short enough, the Exec command **FIX** will load the buffer with the event's input, rather than calling the structure editor. If you really wanted the Lisp editor for your fix, you can say **FIX EVENT - |TTY:|**.

?= Handler

Typing the characters `?=<cr>` displays the arguments to the function currently in progress. Since TTYIN wants you to be able to continue editing the buffer after a `?=`, it prints the arguments below your type-in and then puts the cursor back where it was when `?=` was typed.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. In Interlisp-D, the flags are all initially set to **T**.

?ACTIVATEFLG [Variable]

If true, enables the feature whereby `?` lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis, TTYIN will briefly move the cursor to the matching parenthesis, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you will never notice it).

USERWORDS [Variable]

USERWORDS contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "ED(xx\$)") or type a call to it. If there is no completion for the current word from **USERWORDS**, or there is more than one possible completion, TTYIN beeps. If typed when not inside a word, Escape completes to the value of **LASTWORD**, i.e., the last thing you typed that the Exec noticed, except that Escape at the beginning of the line is left alone (it is an Old Interlisp Exec command).

If you really wanted to enter an escape, you can, of course, just quote it with a CONTROL-V, like you can other control characters.

You may explicitly add words to **USERWORDS** yourself that would not get there otherwise. To make this convenient online the edit command `[←]` means "add the current atom to **USERWORDS**" (you might think of the command as pointing out this atom). For

example, you might be entering a function definition and want to point to one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from **USERWORDS**.

Note that this feature loses some of its value if the spelling list is too long, if there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list **USERWORDS** keeps the temporary section (which is where everything goes initially unless you say otherwise) limited to **#USERWORDS** atoms, initially 100. Words fall off the end if they haven't been used (they are used if **FIXSPELL** corrects to one, or you use <escape> to complete one).

[This page intentionally left blank]

SEdit is the Lisp structure editor. It allows you to edit Lisp code directly in memory. This editor replaces DEdit in Chapter 16, Structure Editor, of the *Interlisp-D Reference Manual*. First introduced in Lyric, the SEdit structure editor has been greatly enhanced in the Medley release. Medley additions are indicated with revision bars in the right margin.

16.1 SEdit - The Structure Editor

As a structure editor, SEdit alters Lisp code directly in memory. The effect this has on the running system depends on what is being edited.

For Common Lisp definitions, SEdit always edits a copy of the object. For example, with functions, it edits the definition of the function. What the system actually runs is the installed function, either compiled or interpreted. The primary difference between the definition and the installed function is that comment forms are removed from the definition to produce the installed function. The changes made while editing a function will not be installed until the edit session is complete.

For Interlisp functions and macros, SEdit edits the actual structure that will be run. An exception to this is an edit of an EXPR definition of a compiled function. In this case, changes are included and the function is unsaved when the edit session is completed.

SEdit edits all other structures, such as variables and property lists, directly. SEdit installs all changes as they are made.

If an error is made during an SEdit session, abort the edit with an Abort command (see Section 16.1.7, Command Keys). This command undoes all changes from the beginning of the edit session and exits from SEdit without changing your environment.

If the definition being edited is redefined while the edit window is open, SEdit redisplay the new definition. Any edits on the old definition will be lost. If SEdit was busy when the redefinition occurred, the SEdit window will be gray. When SEdit is no longer busy, position the cursor in the SEdit window and press the left mouse button; SEdit will get the new definition and display it.

16.1.1 An Edit Session

The List Structure Editor discussion in Chapter 3, Language Integration, explains how to start an editor in Lisp.

Whenever you call SEdit, a new SEdit window is created. This SEdit window has its own process, and thus does not rely on an

Exec to run in. You can make edits in the window, shrink it while you do something else, expand it and edit some more, and finally close the window when you are done.

Throughout an edit session, SEdit remembers everything that you do through a change history. All edits can be undone and redone sequentially. When an edit session ends, SEdit forgets this information and installs the changes in the system.

The session ends with an event signalling to the editor that changes are complete. Three events signal completion:

- Closing the window.

Do this to terminate the edit session when you are finished.

- Shrinking the window.

Shrink the window when you have made some edits and may want to continue the editing session at a later time.

- Typing one of the Completion Commands, listed below.


Each of these commands has the effect of installing your changes, completing the edit, and returning the TTY process to the Exec. They vary in what is done in addition to completing. Using these commands the definition that you were editing can be automatically compiled, the edit window can be closed, or both.

A new edit session begins when you come back to an SEdit after completing. The change history is discarded at this point.

If the Exec is waiting for SEdit to return before going on, complete the edit session using any of the methods above to alert the Exec that SEdit is done. The TTY process passes back to the Exec .

16.1.2 SEdit Carets

There are two carets in SEdit, the edit caret and the structure caret. The edit caret appears when characters are edited within a single structure, such as an atom, string, or comment. Anything typed in will appear at the edit caret as part of the structure that the caret is within. The edit caret looks like this:

(a  b)

The structure caret appears when the edit point is between structures, so that anything inserted will go into a new structure. It looks like this:

(a  b)

SEdit changes the caret frequently, depending on where you are in the structure you are editing, and how the caret is positioned. The left mouse button allows an edit caret position to be set. The middle mouse button allows the structure caret position to be set .

16.1.3 The Mouse

In SEdit, the mouse buttons are used as follows. The left mouse button positions the mouse cursor to point to parts of Lisp structures. The middle mouse button positions the mouse cursor to point to whole Lisp structures. Thus, selecting the Q in LEQ using the left mouse button selects that character, and sets the edit caret after the Q:

```
(LEQ^ n 1)
```

Any characters typed in at this point would be appended to the atom LEQ.

Selecting the same letter using the middle mouse button selects the whole atom (this convention matches TEdit's character/word selection convention), and sets a structure caret between the LEQ and the n:

```
(LEQ^ n 1)
```

At this point, any characters typed in would form a new atom between the LEQ and the n.

Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the desired list to select that list. Press the mouse button multiple times, without moving the mouse, extends the selection. Using the previous example, if the middle button were pressed twice, the list (LEQ ...) would be selected:

```
(LEQ n 1)
```

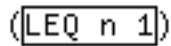
Pressing the button a third time would cause the list containing the (LEQ n 1) to be selected.

The right mouse button positions the mouse cursor for selecting sequences of structures or substructures. Extended selections are indicated by a box enclosing the structures selected. The selection is extended in the same mode as the original selection. That is, if the original selection were a character selection, the right button could be used to select more characters in the same atom. Extended selections also have the property of being marked for pending deletion. That is, the selection takes the place of the caret, and anything typed in is inserted in place of the selection.

For example, selecting the E by pressing the left mouse button and selecting the Q by pressing the right mouse button would produce:

```
(LEQ n 1)
```

Similarly, pressing the middle mouse button and then selecting with the right mouse button extends the selection by whole structures. Thus, in our example, pressing the middle mouse button to select LEQ and pressing the right mouse button to select the 1 would produce:



This is not the same as selecting the entire list, as above. Instead, the elements in the list are collectively selected, but the list itself is not.

16.1.4 Gaps ---

The SEdit structure editor requires that everything edited must have an underlying Lisp structure, even if the structure is not directly displayed. For example, with quoted forms the actual structure might be (**QUOTE** GREEN), although this would be displayed as 'GREEN. Even when the user is in the midst of typing in a form, the underlying Lisp structure must exist.

Because of this necessity, SEdit provides gaps to serve as dummy Lisp objects during typing. SEdit does not need a gap for every form typed in, but gaps are necessary for quoted objects. When something is typed that requires SEdit to build a Lisp structure and thus create a gap, as the quote character does, the gap will appear marked for pending deletion. This means it is ready to be replaced by the structure to be typed in. In this way it is possible to type special structures, like quotes, directly, while SEdit maintains the structure.

A gap looks like: -x-

A gap displayed after a quote has been typed in would look like this:



with the gap marked for pending deletion, ready for typein of the object to be quoted.

16.1.5 Broken Atoms ---

When you are typing an atom (a symbol or a number), SEdit saves the characters you type until you finish the atom. SEdit determines that you've finished the atom when you type a character that cannot (without being escaped) belong to an atom, such as a space or open parenthesis. SEdit then tries to create an atom with these characters, just as if it were the Lisp reader. If it succeeds, the atom becomes part of the structure you're editing. However, if it fails, SEdit intercepts the reader error that would otherwise occur and instead creates a special SEdit structure called a Broken-Atom. A Broken-Atom looks and behaves in SEdit just like a normal atom, but is printed in italics to alert you to its needing correction.

SEdit has to create a Broken-Atom when the characters typed don't make a legal atom. For example, the characters "DECLARE:" cannot make a symbol because the colon is a package specifier, but the form is not correct for a package-qualified symbol. Similarly, the characters "#b123" cannot represent an integer in base two, because 2 and 3 are not legal digits in base two, so SEdit would make a Broken-Atom that looks like *#b123*.

Broken-Atoms can be edited in SEdit just like real atoms. Whenever you finish editing a Broken-Atom, SEdit again tries to create an atom from the characters. If it succeeds, it reprints the atom in SEdit's default font, rather than in italics. You should be sure to correct any Broken-Atoms you create before exiting SEdit, since Broken-Atoms do not behave in any useful way outside SEdit.

16.1.6 Special Characters

A few characters have special meaning in Lisp, and are treated specially by SEdit. SEdit must always have a complete structure to work on at any level of the edit. This means that SEdit needs a special way to type in structures such as lists, strings, and quoted objects. In most instances these structures can be typed in just as they would be to a regular Exec, but in a few cases this is not possible.

Lists- (and)

Lists begin with an open parenthesis character (. Typing an open parenthesis gives a balanced list, that is, SEdit inserts both an open and a close parenthesis. The structure caret is between the two parentheses. List elements can be typed in at the structure caret. When a close parenthesis,) is typed, the caret will be moved outside the list (and the close parenthesis), effectively finishing the list. Square bracket characters, [and], have no special meaning in SEdit, as they have no special meaning in Common Lisp.

Quoted Structures:

SEdit handles the quote keys so that it is possible to type in all quote forms directly. When typing one of the following quote keys at a structure caret, the quote character typed will appear, followed by a gap to be replaced by the object to be quoted.

Single Quote – ’

Use to enter quoted structures.

Backquote – `

Use to enter backquoted structures.

Comma – ,

Use to enter comma forms, as used with a Backquote form.

At Sign – @

Use after a comma to create a comma-at-sign gap. This allows type-in of comma-at forms, e.g. ,@list, as used within a Backquote form.

Dot – .

Use the dot (period) after a comma to create a comma-dot gap. This allows type-in of comma-dot forms, e.g. ,.list, as used within a Backquote form.

Hash Quote – #'

Use this two character sequence to enter the **CL:FUNCTION** abbreviation hash-quote (#').

Dotted Lists:

The dot, or period, character (.) is used to type dotted lists in SEdit. After typing a dot, SEdit inserts a dot and a gap to fill in for the tail of the list. To dot an existing list, point the cursor between the last and second to the last element in the list, and type a dot. To undot a list, select the tail of the list before the dot while holding down the SHIFT key.

Escape- \ or %

Use to escape from a specific typed in character. Use the escape key to enter characters, like parentheses, which otherwise have special meaning to the SEdit reader. Press the escape key then type in the character to escape. SEdit uses the escape key

appropriate to the environment it is editing in; it depends on the readtable that was current when the editor was started. The backslash key (\) is used when editing Common Lisp, and the percent key (%) is used when editing Interlisp.

Multiple Escape- |

Use the multiple escape key, the vertical bar character (|), to escape a sequence of typed in characters. SEdit always balances multiple escape characters. When one multiple escape character is typed, SEdit produces a balanced pair, with the caret between them, ready for typing in the characters to be escaped. If you type a second vertical bar, the caret moves after the second vertical bar, and is still within the same atom, so that you can add more unescaped characters to the atom.

Comments- ;

The comment key, a semicolon (;), starts a comment. When a semicolon is typed, an empty comment is inserted with the caret in position for typing in the comment. Comments can be edited like strings. There are three levels of comments supported by SEdit: single, double, and triple. Single semicolon comments are formatted at the comment column, about three-quarters of the way across the SEdit window, towards the right margin. Double semicolon comments are formatted at the current indentation of the code that they are in. Triple semicolon comments are formatted against the left margin of the SEdit window. The level of a comment can be increased or decreased by pointing after the semicolon, and either typing another semicolon, or backspacing over the preceding semicolon. Comments can be placed anywhere in your Common Lisp code. However, in Interlisp code, they must follow the placement rules for Interlisp comments.

Strings- "

Enter strings in SEdit by typing a double quote ("). SEdit balances the double quotes. When one is typed, SEdit produces a second, with the caret between the two, ready for typing the characters of the string. If a double quote character is typed in the middle of a string, SEdit breaks the string into two smaller strings, leaving the caret between them.

16.1.7 Commands

SEdit commands are most easily entered through the keyboard. When possible, SEdit uses a named key on the keyboard, for example, the DELETE key. The other commands are either Meta, Control, or Meta-Control key combinations. For the alphabetic command keys, either uppercase or lowercase will work.

There are two menus available, as an alternative means of invoking commands. They are the middle button popup menu, and the attached command menu. These menus are described in more detail below.

16.1.8 Editing Commands

Redisplay: Control-L		[Editor Command]
	Redisplays the structure being edited.	
Delete Selection: DELETE		[Editor Command]
	Deletes the current selection.	
Delete Word: Control-W		[Editor Command]

Deletes the previous atom or whole structure. If the caret is in the middle of an atom, deletes backward to the beginning of the atom only.

Control-Meta-O

[Editor Command]

Performs a fast edit by calling ED with its CURRENT option.

16.1.9 Completion Commands

Abort: Meta-A

[Editor Command]

Aborts. This command must be confirmed. All changes since the beginning of the edit session are undone, and the edit is closed.

The following commands signal completion of an edit session and install the structure you were editing.

Control-X

[Editor Command]

Signals the system that this edit is complete. The window remains open, though, so the user can see the edit and start editing again directly.

Control-C

[Editor Command]

Signals the system that this edit is complete and compiles the definition being edited. The variable *compile-fn* determines the function to be called to do the compilation. See the Options section below.

Control-Meta-X

[Editor Command]

Signals the system that this edit is complete and closes the window.

Control-Meta-C

[Editor Command]

Signals the system that this edit is complete, compiles the definition being editing, and closes the window.

16.1.10 Undo Commands

Undo: Meta-U or UNDO

[Editor Command]

Undoes the last edit. All changes since the beginning of the edit session are remembered, and can be undone sequentially.

Redo: Meta-R or AGAIN

[Editor Command]

Redoes the edit change that was just undone. Redo only works directly following an Undo. Any number of Undo commands can be sequentially redone.

16.1.11 Find Commands

Find: Meta-F or FIND

[Editor Command]

Finds a specified structure, or sequence of structures. If there is a current selection, SEdit looks for the next occurrence of the selected structure. If there is no selection, SEdit prompts for the structure to find, and searches forward from the position of the

caret. The found structure will be selected, so the Find command can be used to easily find the same structure again.

If a sequence of structures is selected, SEdit will look for the next occurrence of the same sequence. Similarly, when SEdit prompts for the structure to find, you can type a sequence of structures to look for.

The variable `*wrap-search*` controls whether or not SEdit wraps around from the end of the structure being edited and continues searching from the beginning.

Reverse Find: Control-Meta-F[Editor Command]

Finds a specified structure, searching in reverse from the position of the caret.

The variable `*wrap-search*` controls whether or not SEdit wraps around from the beginning of the structure being edited and continues searching from the end.

Find Gap: Meta-N or SKIP-NEXT[Editor Command]

Skips to the next gap in the structure, leaving it selected for pending deletion.

Substitute: Meta-S or SHIFT-FIND[Editor Command]

Substitutes one structure, or sequence of structures, for another structure, or sequence, within the current selection. SEdit prompts you in the SEdit prompt window for the structures to replace, and the structures to replace with.

The selection to substitute within must be a structure selection. To get a structure selection, click with the middle mouse button (not the left), and extend it, if necessary, with the right mouse button. If you begin with the left button, you will get an informational message "Select the structure to substitute within", because the selection was of characters, rather than structures.

Delete Structure: Control-Meta-S[Editor Command]

Removes all occurrences of a structure or sequence of structures within the current selection. SEdit prompts the user in the SEdit prompt window for the structures to delete.

16.1.12 General Commands

Arglist: Meta-H or HELP[Editor Command]

Shows the argument list for the function currently selected, or currently being typed in, in the SEdit prompt window. If the argument list will not fit in the SEdit prompt window, it is displayed in the main Prompt Window.

Convert Comments: Meta-;[Editor Command]

Converts old style comments in the selected structure to new style comments. This converter notices any list that begins with an asterisk (*) in the INTERLISP package (IL:*) as an old style comment. Section 16.1.18, Options, describes the converter options.

Comment Out Selection: Control-Meta-; [Editor Command]

This command puts the contents of a structure selection into a comment. This provides an easy way to "comment out" a chunk of code. The Extract command can be used to reverse this process, returning the comment to the structures contained therein.

Edit: Meta-O [Editor Command]

Edits the definition of the current selection. If the selected name has more than one type of definition, SEdit asks for the type to be edited. If the selection has no definition, a menu pops up. This menu lets the user specify either the type of definition to be created, or no definition if none needs to be created.

Eval: Meta-E [Editor Command]

Evaluates the current selection. If the result is a structure, the inspector is called on it, allowing the user to choose how to look at the result. Otherwise, the result is printed in the SEdit prompt window. The evaluation is done in the process from which the edit session was started. Thus, while editing a function from a break window, evaluations are done in the context of the break.

Expand: Meta-X or EXPAND [Editor Command]

Replaces the current selection with its definition. This command can be used to expand macros and translate CLISP.

Extract: Meta-/ [Editor Command]

Extracts one level of structure from the current selection. If there is no selection, but there is a structure caret, the list containing the caret is used. This command can be used to strip the parentheses off a list, or to unquote a quoted structure, or to replace a comment with the structures contained therein.

Inspect: Meta-I [Editor Command]

Inspect the current selection.

Join: Meta-J [Editor Command]

Joins. This command joins any number of sequential Lisp objects of the same type into one object of that type. Join is supported for atoms, strings, lists, and comments. In addition, SEdit permits joining of a sequence of atoms and strings, since either type can easily be coerced into the other. In this case, the result of the Join will be an atom if the first object in the selection is an atom, otherwise the result will be a string.

Mutate: Meta-Z [Editor Command]

Mutates. This command allows the user to do arbitrary operations on a LISP structure. First select the structure to be mutated (it must be a whole structure, not an extended selection). When the

user presses Meta-Z SEdit prompts for the function to use for mutating. This function is called with the selected structure as its argument, and the structure is replaced with the result of the mutation.

For example, an atom can be put in upper case by selecting the atom and mutating by the function U-CASE. You can replace a structure with its value by selecting it and mutating by EVAL.

Quote: Meta-'
Meta-'
Meta-,
Meta-.
Meta-@ or Meta-2
Meta-# or Meta-3 [Editor Command]

Quotes the current selection with the specified kind of quote, respectively, Single Quote, Backquote, Comma, Comma-At-Sign, Comma-Dot, or Hash-Quote.

Normalize Selection: Meta-Space or Meta-Return [Editor Command]

Scrolls the current selection to the center of the window. Similarly, the Space or Return key can be used to normalize the caret.

Parenthesize: Meta-) or Meta-0 [Editor Command]

Parenthesizes the current selection, positioning the caret after the new list.

Parenthesize: Meta- (or Meta-9 [Editor Command]

Parenthesizes the current selection, positioning the caret at the beginning of the new list. Only a whole structure selection or an extended selection of a sequence of whole structures can be parenthesized.

16.1.13 Miscellaneous

Change Print Base: Meta-B

[Editor Command]

Changes Print Base. Prompts for entry of the desired Print Base, in decimal. SEdit redisplay fixed point numbers in this new base.

Set Package: Meta-P

[Editor Command]

Changes the current package for this edit. Prompts the user, in the SEdit prompt window, for a new package name. SEdit will redisplay atoms with respect to that package.

Attached Menu: Meta-M

[Editor Command]

Attaches a menu of the commonly used commands (the SEdit Command Menu) to the top of the SEdit window. Each SEdit window can have its own menu, if desired.

16.1.14 Help Menu

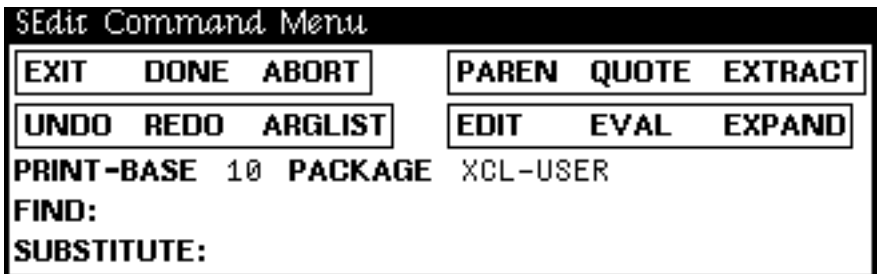
When the mouse cursor is positioned in the SEdit title bar and the middle mouse button is pressed, a Help Menu of commands pops up. The menu looks like this:

Commands	
Abort	M-A
Done	C-X
Done & Compile	C-C
Done & Close	M-C-X
Done, Compile, & Close	M-C-C
Undo	M-U
Redo	M-R
Find	M-F
Reverse Find	M-C-F
Remove	M-C-S
Substitute	M-S
Find Gap	M-N
Arglist	M-H
Convert Comment	M-;
Edit	M-O
Eval	M-E
Expand	M-X
Extract	M-/
Inspect	M-I
Join	M-J
Mutate	M-Z
Parenthesize	M-(
Quote	M-'
Set Print-Base	M-B
Set Package	M-P
Attach Menu	M-M

The Help Menu lists each command and its corresponding Command Key. (In the menu, the letter C stands for CONTROL, while M indicates Meta.) The command selected is executed just as if the command had been entered from the keyboard. The menu remembers which command was selected last, and pops up with the mouse cursor next to that same command the next time the menu is used. This provides a very fast way to repeat the same command when using the mouse.

16.1.15 Command Menu

The SEdit Attached Command Menu contains the commonly used commands. Use the Meta-M keyboard command to bring up this menu. The menu can be closed, independently of the SEdit window, when desired. The menu looks like:



All of the commands in the menu function identically to their corresponding keyboard commands, except for Find and Substitute.

When Find is selected with the mouse cursor, SEdit prompts in the menu window, next to the Find button, for the structures to find. Type in the structures then select Find again. The search begins from the caret position in the SEdit window.

Similarly, Substitute prompts, next to the Find button, for the structures to find, and next to the Substitute button for the structures to substitute with. After both have been typed in, selecting Substitute replaces all occurrences of the Find structures with the Substitute structures, within the current selection.

To do a confirmed substitute, set the edit point before the first desired substitution, and select Find. Then if you want to substitute that occurrence of the structure, select Substitute. Otherwise, select Find again to go on.

Selecting either Find or Substitute with the right mouse button erases the old structure to find or substitute from the menu, and prompts for a new one.

16.1.16 SEdit Programmer's Interface

The following sections describe SEdit's programmer's interface. All symbols are external in the package named "SEdit".

16.1.17 SEdit Window Region Manager

SEdit provides user redefinable functions which control how SEdit chooses the region for a new edit window.

(get-window-region *context reason name type*)

[Function]

This function is called when SEdit wants to know where to place a window it is about to open. This happens whenever the user starts a new SEdit or expands an Sedit icon. The default behavior is to pop a window region off SEdit's stack of regions that have been used in the past. If the stack is empty, SEdit prompts for a new region.

This function can be redefined to provide different behavior. It is called with the edit *context*, a *reason* for needing a region, the *name* of the structure to be edited, and the *type* of the structure to be edited. The edit *context* is SEdit's main data structure and can be useful for associating particular edits with specific regions. The *reason* argument specifies why SEdit wants a region, and will be one of the keywords :CREATE or :EXPAND.

(save-window-region *context reason name type region*)

[Function]

This function is called whenever SEdit is finished with a region and wants to make the region available for other SEdits. This happens whenever an SEdit window is closed or shrunk, or when an SEdit Icon is closed. The default behavior is simply to push the region onto SEdit's stack of regions.

This function can be redefined to provide different behavior. It is also called with the edit *context*, the *reason*, the *name*, the *type*, and additionally the window *region* that is being released. The *reason* argument specifies why SEdit is releasing the region, and will be one of the keywords :CLOSE, :SHRINK, or :CLOSE-ICON.

keep-window-region

[Variable]

Default **T**. This flag determines the behavior of the default SEdit region manager, explained above, for shrinking and expanding windows. When set to **T**, shrinking an SEdit window will not give up that window's region; the icon will always expand back into the same region. When set to **NIL**, the window's region is made available for other SEdits when the window is shrunk. Then when an SEdit icon is expanded, the window will be reshaped to the next available region.

This variable is only used by the default implementations of the functions **get-window-region** and **save-window-region**. If these functions are redefined, this flag is no longer used.

16.1.18 Options

The following parameters can be set as desired.

wrap-parens

[Variable]

This SEdit pretty printer flag determines whether or not trailing close parenthesis characters, `)`, are forced to be visible in the window without scrolling. By default it is set to **NIL**, meaning that close parens are allowed to "fall off" the right edge of the window. If set to **T**, the pretty printer will start a new line before the structure preceding the close parens, so that all the parens will be visible.

wrap-search

[Variable]

This flag determines whether or not SEdit find will wrap around to the top of the structure when it reaches the end, or vice versa in the case of reverse find. The default is NIL.

clear-linear-on-completion [Variable]

This flag determines whether or not SEdit completely re-pretty prints the structure being edited when you complete the edit. The default value is NIL, meaning that SEdit reuses the pretty printing.

ignore-changes-on-completion [Variable]

Sometimes the structure that you are editing is changed by the system upon completion. Editdates are an example of this behavior. When this flag is NIL, the default, SEdit will redisplay the new structure, capturing the changes. When T, SEdit will ignore the fact that changes were made by the system and keep the old structure.

convert-upgrade [Variable]

Default 100. When using Meta-; to convert old-style single- asterisk comments, if the length of the comment exceeds **convert-upgrade** characters, the comment is converted into a double semicolon comment. Otherwise, the comment is converted into a single semicolon comment.

Old-style double-asterisk comments are always converted into new-style triple-semicolon comments.

16.1.19 Control Functions

(reset) [Function]

This function recomputes the SEdit edit environment. Any changes made in the font profile, or any changes made to SEdit's commands are captured by resetting. Close all SEdit windows before calling this function.

(add-command *key-code form &optional scroll? key-name command-name help-string*)

[Function]

This function allows you to write your own SEdit keyboard commands. You can add commands to new keys, or you can redefine keys that SEdit already uses as command keys. If you mistakenly redefine an SEdit command, the function Reset-Commands will remove all user-added commands, leaving SEdit with its default set of commands.

key-code can be a character code, or any form acceptable to il:charcode.

form determines the function to be called when the key command is typed. It can be a symbol naming a function, or a list, whose first element is a symbol naming a function and the rest of the elements are extra arguments to the function. When the command is invoked, SEdit will apply the function to the edit context (SEdit's main data structure), the charcode that was typed, and any extra arguments supplied in *form*. The extra arguments do not get evaluated, but are useful as keywords or flags, depending on how the command was invoked. The command function must return T if

it handled the command. If the function returns NIL, SEdit will ignore the command and insert the character typed.

The first optional argument, *scroll?*, determines whether or not SEdit scrolls the window after running the command. This argument defaults to NIL, meaning don't scroll. If the value of SCROLL is T, then SEdit will scroll the window to ensure that the caret is visible.

The rest of the optional arguments are used to add this command to SEdit's middle button menu. When the item is selected from the menu, the command function will be called as described above, with the *charcode* argument set to NIL.

key-name is a string to identify the key (combination) to be typed to invoke the command. For example "M-A" to represent the Meta-A key combination, and "C-M-A" for Control-Meta-A.

command-name is a string to identify the command function, and will appear in the menu next to the *key-name*.

help-string is a string to be printed in the prompt window when a mouse button is held down over the menu item.

After adding all the commands that you want, you must call Reset-Commands to install them.

For example:

```
(add-command "^U" (my-change-case t))
(add-command "^Y" (my-change-case nil))
(add-command "l,r" my-remove-nil
  "M-R" "Remove NIL"
  "Remove NIL from the selected structure"))
(reset-commands)
```

will add three commands. Suppose *my-change-case* takes the arguments *context*, *charcode*, and *upper-case?*. *upper-case?* will be set to T when *my-change-case* is called from Control-U, and NIL when called from Control-Y. *my-remove-nil* will be called with only *context* and *charcode* arguments when Meta-R is typed.

Below are some SEdit functions which are useful in writing new commands.

(reset-commands)

[Function]

This function installs all commands added by **add-command**. SEdits which are open at the time of the **reset-commands** will not see the new commands; only new SEdits will have the new commands available.

(default-commands)

[Function]

This function removes all commands added by **add-command**, leaving SEdit with its default set of commands. As in **reset-commands**, open SEdits will not be changed; only new SEdits will have the user commands removed.

(get-prompt-window *context*)**[Function]**

This function returns the attached prompt window for a particular SEdit.

(get-selection *context*)**[Function]**

This function returns two values: the selected structure, and the type of selection, one of NIL, T, or :SUB-LIST. The selection type NIL means there is not a valid selection (in this case the structure is meaningless). T means the selection is one complete structure. :SUB-LIST means a series of elements in a list is selected, in which case the structure returned is a list of the elements selected.

(replace-selection *context structure selection-type*)**[Function]**

This function replaces the current selection with a new structure, or multiple structures, by deleting the selection and then inserting the new structure(s). The *selection-type* argument must be one of T or :SUB-LIST. If T the *structure* is inserted as one complete structure. If :SUB-LIST, the *structure* is treated as a list of elements, each of which is inserted.

edit-fn**[Variable]**

This function is funcalled with the selected structure and the edit options as its arguments from the Edit (M-O) command. It should start the editor as appropriate, or else generate an error if the selection is not editable.

compile-fn**[Variable]**

This function is funcalled with the arguments *name*, *type*, and *body*, from the compile completion commands. It should compile the definition, *body*, and install the code as appropriate.

(sedit *structure props options*)**[Function]**

This function provides a means of starting SEdit directly. *structure* is the structure to be edited.

props is a property list, which may specify the following properties:

:name - the name of the object being edited

:type - the file manager type of the object being edited. If NIL, SEdit will not call the file manager when it tries to refetch the definition it is editing. Instead, it will just continue to use the structure that it has.

:completion-fn - the function to be called when the edit session is completed. This function is called with the *context*, *structure*, and *changed?* arguments. *context* is SEdits main data structure. *structure* is the structure being edited. *changed?* specifies if any changes have been made, and is one of NIL, T, or :ABORT, where :ABORT means the user is aborting the edit and throwing away any changes made. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the main arguments above.

:root-changed-fn - the function to be called when the entire structure being edited is replaced with a new structure. This function is called with the new structure as its argument. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the structure argument.

options is one or a list of any number of the following keywords:

:close-on-completion - This option specifies that SEdit cannot remain active for multiple completions. That is, the SEdit window cannot be shrunk, and the completion commands that normally leave the window open will in this case close the window and terminate the edit.

:compile-on-completion - This option specifies that SEdit should call the *compile-fn* to compile the definition being edited upon completion, regardless of the completion command used.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by SEdit or the XCL compiler. It is possible to insert a comment at the beginning of your function that looks like

```
(* DECLARATIONS: --)
```

The tail, or -- section, of this comment is taken as a set of local record declarations which are then used by the compiler in that function just as if they had been declared globally. See the "Compiler" section in Chapter 3 of these Notes for additional behavior in XCL.

SEdit does not recognize such declarations. Thus, if the "Expand" command is used, the expansion will not be done with these record declarations in effect. The code that you see in SEdit will not be the same code compiled by the BYTECOMPILER.

[This page intentionally left blank]

ICONW, used to build small windows that will appear as icons on the display, is a standard input/output feature. This feature was introduced in Lyric and has been enhanced in Medley. The following description of **ICONW** should be appended to Section 28.4, Windows, of the *Interlisp-D Reference Manual*. Medley changes are indicated with revision bars in the right margin.

28.4.16 Creating Icons with ICONW

ICONW is a group of functions available for building small windows of arbitrary shape. These windows are principally for use as icons for shrinking windows; i.e., these functions are likely to be invoked from within the **ICONFN** of a window. An icon is specified by supplying its image (a bitmap) and a mask that specifies its shape. The mask is a bitmap of the same dimensions as the image whose bits are on (black) in those positions considered to be in the image, and off (white) in those positions where the background should show through. By using the mask and appropriate window functions, **ICONW** maintains the illusion that the icon window is nonrectangular, even though the actual window itself is rectangular. The illusion is not complete, of course. For example, if you try to select what looks like the background (or an occluded window) around the icon but still within its rectangular perimeter, the icon window itself is selected. Also, if you move a window occluded by an icon, the icon never notices that the background changed behind it. Icons created with **ICONW** can also have titles; some part of the image can be filled with text computed at the time the icon is created, or text may be changed after creation.

28.4.16.1 Creating Icons

Two types of icons can be created with **ICONW**, a borderless window containing an image defined by a mask and a window with a title.

(ICONW IMAGE MASK POSITION NOOPENFLG) [Function]

Creates a window at *POSITION*, or prompts for a position if *POSITION* is **NIL**. The window is borderless, and filled with *IMAGE*, as cookie-cut by *MASK*. If *MASK* is **NIL**, the image is considered rectangular (i.e., *MASK* defaults to a black bitmap of the same dimensions as *IMAGE*). If *NOOPENFLG* is **T**, the window is returned unopened.

(TITLEDICONW ICON TITLE FONT POSITION NOOPENFLG JUST BREAKCHARS OPERATION) [Function]

Creates a titled icon at *POSITION*, or prompts for a position if *POSITION* is **NIL**. If *NOOPENFLG* is **T**, the window is returned unopened. The argument *ICON* is an instance of the record **TITLEDICON**, which specifies the icon image and mask, as with **ICONW**, and a region within the image to be used for displaying the title. Thus, the *ICON* argument is usually of the form

```
(create TITLEDICON ICON ← someIconImage
```



```
MASK ← iconMask  TITLEREG ←  
someRegionWithinICON)
```

The title region is specified in coordinates relative to the icon, i.e., the lower-left corner of the image bitmap is (0, 0). The mask can be **NIL** if the icon is rectangular. The image should be white where it is covered by the title region. **TITLEDICONW** clears the region before printing on it. The title is printed into the specified region in the image, using *FONT*. If *FONT* is **NIL** it defaults to the value of **DEFAULTICONFONT**, initially Helvetica 10. The title is broken into multiple lines if necessary; **TITLEDICONW** attempts to place the breaks at characters that are in the list of character codes *BREAKCHARS*. *BREAKCHARS* defaults to (**CHARCODE (SPACE** *ȳ* **)**). In addition, line breaks are forced by any carriage returns in *TITLE*, independent of *BREAKCHARS*. *BREAKCHARS* is ignored if a long title would not otherwise fit in the specified region. For convenience, *BREAKCHARS* = **FILE** means the title is a file name, so break at file name field delimiters. The argument *JUST* indicates how the text should be justified relative to the region. It is an atom or list of atoms chosen from **TOP**, **BOTTOM**, **LEFT**, or **RIGHT**, which indicate the vertical positioning (flush to top or bottom) and/or horizontal positioning (flush to left edge or right). If *JUST* = **NIL**, the text is centered. The argument *OPERATION* is a display stream operation indicating how the title should be printed. If *OPERATION* is **INVERT**, then the title is printed white-on-black. The default *OPERATION* is **REPLACE**, meaning black-on-white. **ERASE** is the same as **INVERT**; **PAINT** is the same as **REPLACE**.

For convenience, **TITLEDICONW** can also be used to create icons that consist solely of a title, with no special image. If the argument *ICON* is **NIL**, **TITLEDICONW** creates a rectangular icon large enough to contain *TITLE*, with a border the same width as that on a regular window. The remaining arguments are as described above, except that a *JUST* of **TOP** or **BOTTOM** is not meaningful.

In the Medley release, **TITLEDICONW** can create icons with white text on a black background. To get this effect, your icon image must be black in the correct area, and you must specify the *OPERATION* argument as **INVERT**.

In Medley, you can copy- select the title of an icon.

28.4.16.2 Modifying Icons

(ICONW.TITLE <i>ICON TITLE</i>)	[Function]
--	------------

Returns the current title of the window *ICON*, which must be a window returned by **TITLEDICONW**. In addition, if *TITLE* is non-**NIL**, makes *TITLE* the new title of the window and repaints it accordingly. To erase the current title, make *TITLE* a null string.

(ICONW.SHADE <i>WINDOW SHADE</i>)	[Function]
--	------------

Returns the current shading of the window *ICON*, which must be a window returned by **ICONW** or **TITLEDICONW**. In addition, if *SHADE* is non-**NIL**, paints the texture *SHADE* on *WINDOW*. A typical use for this function is to communicate a change of state in a window that is shrunk, without reopening the window. To remove any shading, make *SHADE* be **WHITESHADE**.

28.4.16.3 Default Icons

When you shrink a window that has no **ICONFN**, the system currently creates an icon that looks like the window's title bar. You can make the system instead create titled icons by setting the global variable **DEFAULTICONFN** to the value **TEXTICON**.

(TEXTICON WINDOW TEXT) [Function]

Creates a titled icon window for the main window *WINDOW* containing the text *TEXT*, or the window's title if *TEXT* is **NIL**.

DEFAULTTEXTICON [Variable]

The value that **TEXTICON** passes to **TITLEDICONW** as its *ICON* argument. Initially it is **NIL**, which creates an unadorned rectangular window. However, you can set it to a **TITLEDICON** record of your choosing if you would like default icons to have a different appearance.

28.4.16.4 Sample Icons

The LispUsers StockIcons module contains a collection of icons and their masks usable with **ICONW**, including:

- **FOLDER, FOLDERMASK** - a file folder
- **PAPERICON, PAPERICONMASK** - a sheet of paper with the top right corner turned
- **FILEDRAWER, FILEDRAWERMASK** - front of a file drawer
- **ENVELOPEICON, ENVELOPEMASK** - envelope
- **TITLED.FILEDRAWER** - TitledIcon of the filedrawer front (capacity, about three lines of 10-point text)
- **TITLED.FILEFOLDER** - TitledIcon of the file folder (capacity, about three lines of 10-point text)
- **TITLED.ENVELOPE** - TitledIcon of the envelope (capacity, one short line of 10-point text)

[This page intentionally left blank]

APPENDIX E. ERROR SYSTEM

This appendix replaces Chapter 24, Error System, of *Common Lisp Implementation Notes*, Lyric Release, which replaced most of Chapter 24, Errors, of *Common Lisp, the Language*. Text shown with ~~StrikeThru~~ is that text from the Lyric release that no longer applies in Medley. Enhancements added in Medley are indicated with revision bars in the right margin.

The XCL error system has been updated to reflect the current ANSI Common Lisp error system proposal. This version seems to be gaining wide use in other Common Lisp implementations, so no further major changes are anticipated.

The Common Lisp error system is based on proposal number 18 for the Common Lisp error system. Deviations from this proposal are noted. Since the Common Lisp error system has not yet been standardized, this system may change in future releases to accommodate the final version of the Common Lisp error system.

If you have access to the ARPANet, a copy of this proposal may be retrieved from MIT-AI.ARPA as the file "COMMON;COND18 TXT".

All symbols described in the error system proposal that are not already in the "LISP" package are exported from the "CONDITIONS" package. In addition, the "XEROX-COMMON-LISP" package exports these symbols, so you can make them available either by using "XCL" or using "CONDITIONS", whichever is appropriate to your application. The distinction is made so that XCL extensions of the Common Lisp error system will be clear. All unqualified symbols are assumed to be in the "LISP" package.

Summary of Error System Changes

The semantics of HANDLER-BIND where multiple bindings are set up or mutple condition types are being handled are slightly different. Old code that used this will probably not behave as expected.

HANDLER-BIND and HANDLER-CASE (a.k.a. CONDITION-CASE) now always take a typespec instead of a list of condition types to indicate the conditions to be handled. Old code that uses this will only handle the first condition type in the list. The function, CONDITIONS::CONVERT-HANDLER-CASE is provided to aid in converting old code. It may be used as a mutation function in SEdit.

HANDLER-CASE now supports a :NO-ERROR option that is executed if none of the other clauses are taken. This is handy for writing code that depends on the normal completion of some operation, for example, creating auxilliary files if a particular stream is successfully opened.

SERIOUS-CONDITION no longer forces entry to the debugger. The function used to signal the condition now determines what happens if the condition is not handled. This means that SERIOUS-CONDITION has no more interesting properties and is likely to be removed in the final version of the error standard.

Several new condition types have been defined. Others have moved in the hierarchy. For example, ILLEGAL-GO is now a subtype of PROGRAM-ERROR.

No standard condition type has a default handler.

The standard debugger entry point is now called INVOKE-DEBUGGER instead of DEBUG.

The syntax of DEFINE-CONDITION has been changed to make it more like CLOS' DEFCLASS. The function CONDITIONS::CONVERT-OLD-DEFINE-CONDITION is provided to aid in converting old code. It may be used as a mutation function in SEdit.

Several DEFINE-CONDITION options have been merged, while others have been removed. In particular, there are no more "instant variables."

PROCEED-CASE has been replaced by RESTART-CASE. The semantics of restarts have been cleaned up and several new features added. Related functions, such as COMPUTE-PROCEED-CASES, have been renamed appropriately.

INVOKE-PROCEED-CASE has been renamed to INVOKE-RESTART.

DEFINE-PROCEED-FUNCTION has been removed, although XCL will continue to support it for compatibility.

The arguments to a restart's report function are different. Old code that used something other than a string for the report method will not work correctly.

A distinction is now made between invoking a restart interactively and simply invoking one. To this end, there is the function INVOKE-RESTART-INTERACTIVELY and the :INTERACTIVE option to RESTART-CASE.

RESTART-BIND, in analogy to HANDLER-BIND, has been added.

A new variable, *BREAK-ON-SIGNALS* exists to aid in debugging. It is a generalization of *BREAK-ON-WARNINGS*. The latter has been retained for compatibility.

The proceed function PROCEED has been changed to CONTINUE.

Old compiled code will continue to work except in the following cases, some of which have been mentioned above:

A proceed case's report function was not a simple string. Such code can cause stack overflow trying to report the condition (*STANDARD-OUTPUT* ends up being bound to NIL). Such code should be rewritten.

A handler binding is made to a list of condition types. Only the first type in the list will be handled.

Multiple handler bindings were created by the same HANDLER-BIND or HANDLER-CASE. Such code will work as expected, but if recompiled in Medley, will not. To get the effect of the current semantics, you must use nested HANDLER-BINDs.

Under the new error system, `use-value` and `store-value` no longer prompt for a value.

Introduction to Error System Terminology

condition A *condition* is a kind of object which is created when an exceptional situation arises in order to represent the relevant features of that situation.

signal, handlers Once a condition is created, it is common to *signal* it. When a condition is signaled, a set of *handlers* are tried in some pre-defined order until one decides to *handle* the condition or until no more handlers are found. A condition is said to have been handled if a handler performs a non-local transfer of control to exit the signalling process.

restart Although such transfers of control may be done directly using traditional Lisp mechanisms such as `catch` and `throw`, `block` and `return`, or `tagbody` and `go`, the condition system also provides a more structured way to *restart* a computation. Among other things, the use of these structured primitives for restarting allows a better and more integrated relationship between the user program and the interactive debugger.

~~*serious conditions* It is not necessary that all conditions be handled. Some conditions are trivial enough that a failure to handle them may be disregarded. Others, which we will call *serious conditions* must be handled in order to assure correct program behavior. If a serious condition is signalled but no handler is found, the debugger will be entered so that the user may interactively specify how to proceed.~~

errors conditions which result from incorrect programs or data are called *errors*. Not all conditions are errors, however. Storage conditions are examples of conditions that are not errors. For example, the control stack may legitimately overflow without a program being in error. Even though a stack overflow is not necessarily a program error, it is serious enough to warrant entry to the debugger if the condition goes unhandled.

Some types of conditions are predefined by the system. All types of conditions are subtypes of `conditions:condition`. That is,

```
(typep c 'conditions:condition)
```

is true if `c` is a condition.

creating conditions The only standard way to define a new condition type is `conditions:define-condition`. The only standard way to instantiate a condition is `conditions:make-condition`.

When a condition object is created, the most common operation to be performed upon it is to *signal* it (although there may be applications in which this does not happen, or does not happen immediately).

When a condition is signaled, the system tries to locate the most appropriate handler for the condition and invoke that handler. Handlers are located according to the following rules:

- bound*
- Check for locally defined (ie, *bound*) handlers.
 - If no appropriate bound handler is found, check first for the default handler of the signaled type and then of each of its superiors.

decline If an appropriate handler is found, the handler may *decline* by simply returning without performing a non-local transfer of control. In such cases, the search for an appropriate handler is picked up where it left off, as if the called handler had never been present. When a handler is running, the "handler binding stack" is popped back to just below the binding that caused that handler to be invoked. This is done to avoid infinite recursion in the case that a handler also signals a condition.

`conditions:handler-bind` When a condition is signaled, handlers are searched for in the dynamic environment of the signaller. Handlers can be established within a dynamic context by use of `conditions:handler-bind` and other forms based on it.

handler A *handler* is a function of one argument, the condition to be handled. The handler may inspect the object (using primitives described in another section) to be sure it is interested in handling the condition. After inspecting the condition, the handler must take one of the following actions:

- It may decline to handle the condition by simply returning. When this happens, any returned values are ignored and the effect on the signaling process is the same as if the handler had not run. The next handler in line will be tried, or if no such handler exists, the default action for the given condition will be taken. A default handler may also decline, in which case the condition will go unhandled. What happens then depends on which function was used to signal the condition (`xcl:signal`, `error`, `cerror`, `warn`).
- It may perform some non-local transfer of control using `go`, `return`, `throw`, `abort`, or `conditions:invoke-restart`.
- It may signal another condition.
- It may invoke the debugger.

`conditions:restart-case` When a condition is signalled, a facility is available for use by handlers to transfer control to an outer dynamic contour of the program. The form which creates contours that may be returned to is `conditions:restart-case`. Each contour is set up by a `conditions:restart-case` clause, and is called a *restart*. The function that transfers control to a restart is `conditions:invoke-restart`.

~~*proceed function*~~ Also, control may be transferred along with parameters to a named ~~`xcl:proceed`~~ case clause by invoking a ~~*proceed function*~~ of that name.

~~*Proceed functions*~~ are created with the macro ~~`xcl:define-proceed-function`~~.

restart type A restart with a particular name is sometimes called a *restart type*.

report In some cases, it may be useful to *report* a condition or a restart to a user or a log file of some sort. When the printer is invoked on a condition or proceed case and `*print-escape*` is nil, the report function for that object is invoked. In particular, this means that an expression like

```
(princ condition)
```

will invoke `condition's` report function. Because of this, no special function is provided for invoking the report function of a condition or a restart.

Program Interface to the Condition System

Defining and Creating Conditions

`conditions:define-condition` *name* (*parent-type*) [(*{slot}**) *{option}**]

[Macro]

Defines a new condition type with the given *name*, making it a subtype of the given *parent-type*.

Except as otherwise noted, the arguments are not evaluated.

The valid *options* are:

```
(:documentation doc-string)
```

doc-string should be a string which describes the purpose of the condition type or NIL. If this option is omitted, NIL is assumed. (`documentation name 'type`) will retrieve this information.

```
(:conc-name symbol-or-string)
```

As in `defstruct`, this sets up automatic prefixing of the names of slot accessors. Also as in `defstruct` if no prefix is specified the default behavior for automatic prefixing is to use the name of the new type followed by a hyphen interned in the

package which is current at the time that the `conditions:define-condition` is processed.

~~`:report-function expression`~~

~~expression should be a suitable argument to the function special form, e.g., a symbol or a lambda expression. It designates a function of two arguments, a condition and a stream, which prints the condition to the stream when `*print-escape*` is nil.~~

~~The `:report-function` describes the condition in a human-sensible form. This item is somewhat different than a structure's `:print-function` in that it is only used if `*print-escape*` is nil.~~

`(:report exp)`

This option specifies the report function for this condition type. Report function are inherited, so if a particular condition type does not have one, the report function of its parent will be used.

If *exp* is a string, it is a shorthand for

```
(:report (lambda (condition stream)
          (declare (ignore conditions))
          (princ exp stream)))
```

If *exp* is not a string, `(function exp)` will be evaluated in the current lexical environment. This should return a function of two arguments, a condition and a stream. It will be called when a condition of this type is to be printed and `*print-escape*` is nil. The report function will be called with the condition to be reported and the stream to which the report is to be made.

~~`:handler-function expression`~~

~~expression should be a suitable argument to the function special form. It designates a function of one argument, a condition, which may handle that condition if no dynamically-bound handler did.~~

`(:handle exp)`

This option specifies a default handler for conditions of this type. `(function exp)` will be evaluated in the current lexical context. This should result in a function of one argument, a condition, to be used as the default handler for this condition type.

Each *slot* is a `defstruct slot-description`. In addition to those specified, the slots of the *parent-type* are also available. No slot-options are allowed, only an optional default-value expression. Condition objects are immutable, i.e., all of their slots are automatically declared to be `:read-only`.

conditions:make-condition will accept keywords with the same name as any of the slots, and will initialize the corresponding slots in conditions it creates.

Accessors are created according to the same rules as used by defstruct. For example:

```
(conditions:define-condition bad-food-color (food-lossage)
  (food color)
  (:report (lambda (c s) (format s "The food ~A was ~A"
                                (bad-food-color-food c) (bad-food-
color-color c))))))
```

defines a condition of type bad-food-color which inherits from the food-lossage condition type. The new type has slots food and color so that conditions:make-condition will accept :food and :color keywords and accessors bad-food-color-food and bad-food-color-color will apply to objects of this type.

The report function for a condition will be implicitly called any time a condition is printed with *print-escape* being nil. Hence,

```
(princ condition)
```

is a way to invoke the condition's report function.

Here are some examples of defining condition types. This form defines a condition called machine-error which inherits from error:

```
(conditions:define-condition machine-error (error) (machine-
name)
  (:report (lambda (c s) (format s
                                "There is a problem with ~A."
                                (machine-error-machine-name c)))))
)
```

The following defines a new error condition (a subtype of machine-error) for use when machines are not available:

```
(conditions:define-condition machine-not-available-error
  (machine-error) (machine-name)
  (:report (lambda (c s) (format s
                                "The machine ~A is not available."
                                (machine-error-machine-name c)))))
)
```

The following defines a still more specific condition, built upon machine-not-available-error, which provides a default for machine-name but which does not provide any new slots:

```
(conditions:define-condition
  my-favorite-machine-not-available-error
  (machine-not-available-error)
  ((machine-name "Tesuji:AISDev")))
```

This gives the machine-name slot a default initialization. Since no :report clause was given, the information supplied in the definition of machine-not-available-error will be used if a condition of this type is printed while *print-escape* is nil.

Returns the object used to report conditions of the given *type*. This will be either a string, a function of two arguments (condition and stream) or `nil` if there is no report function. `setf` may be used with this form to change the report function for a condition type.

`xcl:condition-handler` *type* [Macro]

Returns the default handler for conditions of the given *type*. This will be a function of one argument or `nil` if there is no default handler. `setf` may be used with this form to change the default handler for a condition type.

`conditions:make-condition` *type* &rest *slot-initializations* [Function]

Calls the appropriate constructor function for the given *type*, passing along the given slot initializations to the constructor, and returning an instantiated condition.

The *slot-initializations* are given in alternating keyword/value pairs. eg,

```
(conditions:make-condition 'bad-food-color
  :food my-food
  :color my-color)
```

This function is provided mainly for writing subroutines that manufacture a condition to be signaled. Since all of the condition signalling functions can take a *type* and *slot-initializations*, it is usually easier to call them directly.

Signalling Conditions

`xcl:*current-condition*` [Variable]

This variable is bound by condition-signalling forms (`conditions:signal`, `error`, `cerror`, and `warn`) to the condition being signaled. This is especially useful in restart filters. The top-level value of `xcl:*current-condition*` is `nil`.

`conditions:signal` *datum* &rest *arguments* [Function]

Invokes the signal facility on a condition. If the condition is not handled, `conditions:signal` returns the condition object that was signaled.

If *datum* is a condition then that condition is used directly. In this case, it is an error for *arguments* to be non-`nil`.

If *datum* is a condition type, then the condition used is the result of doing

```
(apply #'conditions:make-condition
  datum arguments)
```

If *datum* is a string, then the condition used is the result of doing

```
(conditions:make-condition
  'conditions:simple-condition
  :format-string datum
  :format-arguments arguments).
```

~~If the condition is of type `xcl:serious-condition`, then `xcl:signal` will behave exactly like `error`, i.e., it will call~~

~~xel:debug~~ if the condition isn't handled, and will never return to its caller.

If (typep *condition* conditions:*break-on-signals*) is true, then the debugger will be entered prior to the signalling process. This is true for all other functions and macros that signal conditions, such as warn, error, cerror, assert and check-type.

conditions:*break-on-signals*

[Variable]

This flag is primarily for use when debugging programs that do signaling. Its value is a type specifier.

When (typep *condition* conditions:*break-on-signals*) is true, then calls to conditions:signal and other functions that implicitly call conditions:signal will enter the debugger prior to signalling the condition. The conditions:continue restart may be used to continue with the normal signalling process.

The default value of this variable is nil.

Note: the variable *break-on-warnings* continues to be supported for compatibility, but conditions:*break-on-signals* offers that power and more. New code should not use *break-on-warnings*.

error datum &rest arguments

[Function]

Like conditions:signal except if the condition is not handled, the debugger is called with the given condition, and error never returns.

datum is treated as in conditions:signal. If *datum* is a string, a condition of type conditions:simple-error is made. This form is compatible with that described in Steele's *Common Lisp, the Language*.

ccerror proceed-format-string datum &rest arguments

[Function]

Like error, if the condition is not handled the debugger is called with the given condition. However, cerror enables the restart conditions:continue, which will simply return the condition being signalled from cerror.

datum is treated as in error. If *datum* is a condition, then that condition is used directly. In this case, *arguments* will be used only with the *proceed-format-string* and will not be used to initialize *datum*.

The *proceed-format-string* must be a string. Note that if *datum* is not a string, then the format arguments used by the *proceed-format-string* will still be the *arguments* (in the keyword format as specified). In this case, some care may be necessary to set up the *proceed-format-string* correctly. The format directive ~* may be particularly useful in this situation.

The value returned by cerror is the condition which was signaled.

See Steele's *Common Lisp, the Language*, page 430 for examples of the use of `cerror`.

warn *datum* &**rest** *arguments*

[Function]

Invokes the signal facility on a condition. If the condition is not handled, then the text of the warning is printed on `*error-output*`. If the variable `*break-on-warnings*` is true, then in addition to printing the warning, the debugger is entered using the function `break`. The value returned by `warn` is the condition that was signalled.

If *datum* is a condition, then that condition is used directly. In this case, if the condition is not of type `conditions:warning` or *arguments* is non-null, then an error of type `conditions:type-error` is signalled.

If *datum* is a condition type, then the condition used is the result of doing `(apply #'conditions:make-conditions datum arguments)`. This result must be of type `conditions:warning` or an error of type `conditions:type-error` is signalled.

If *datum* is a string, then the condition used is the result of `(conditions:make-conditions 'conditions:simple-warning :format-string datum :format-arguments arguments)`.

The precise mechanism for warning is as follows:

- 1) If `*break-on-warnings*` is true, the debugger will be entered. This feature is primarily for compatibility with old code: use of `conditions:*break-on-signals*` is preferred. If the break is continued using the `conditions:continue` restart, `warn` proceeds with step 2.
- 2) The warning condition is signalled. While it is being signalled, the `conditions:muffle-warning` restart is established for use by a handler to bypass further action by `warn`, i.e., to cause `warn` to immediately return.
- 3) The warning condition is reported to `*error-output*` by the `warn` function. Note that `warn` will indicate that the condition being signalled is a warning when it reports it, so there is no need for the condition to do so in its report method.

break-on-warnings

[Variable]

check-type

[Macro]

ecase

[Macro]

ccase

[Macro]

etypecase

[Macro]

ctypecase

[Macro]

assert

[Macro]

All of the above behave as described in *Common Lisp: the Language*. The default clauses of `ecase` and `ccase` forms signal `conditions:simple-error` conditions. The default clauses of

`etypecase` and `ctypecase` forms signal `conditions:type-error` conditions. `assert` signals the `xcl:assertion-failed` condition. `ccase` and `ctypecase` set up a `conditions:store-value` restart.

Handling Conditions

`conditions:handler-bind` *bindings* &**rest** *forms* [Macro]

Executes the forms in a dynamic context where the given local handler *bindings* are in effect. The elements of *bindings* must take the form (*type-spec handler*). The handlers are bound in the order they are given, i.e., when searching for a handler, the error system will consider the leftmost binding in a particular `conditions:handler-bind` form first. However, while one of these handlers is running, none of the bindings established by the `conditions:handler-bind` will be in effect.

type must be a type specifier. To make a binding for several condition types, use (*or type1 type2 ...*).

handler should evaluate to a function of one argument, a condition, to be used to handle a signalled condition during execution of the *forms*.

An example of the use of `conditions:handler-bind` appears at the end of the `conditions:restart-case` macro description.

`conditions:handler-case` *form* &**rest** *cases* [Macro]

`xcl:condition-case` *form* &**rest** *cases* [Macro]

Executes the given *form*. Each *case* has the form

(*type* ([*var*]) . *body*)

If a condition is signalled (and not handled by an intervening handler) during the execution of the form, and there is an appropriate clause—i.e., one for which

(*typep condition* ' *type*)

is true—then control is transferred to the body of the relevant clause, binding *var*, if present, to the condition that was signaled. If no condition is signalled, then the values resulting from the *form* are returned by the `xcl:condition-case`. If the condition is not needed, *var* may be omitted.

Earlier clauses will be considered first by the error system. I.e.,

```
(xcl:condition-case form
  (cond1 ...)
  (cond2 ...))
```

is equivalent to

```
(xcl:condition-case
  (xcl:condition-case form
    (cond1 ...))
  (cond2 ...))
```

~~type may also be a list of types, in which case it will catch conditions of any of the specified types.~~

One may also specify an action to be taken if execution of *form* completes normally. This may be done by specifying a clause that has `:no-error` as its type. Such a clause, if provided, must be last. A `:no-error` clause looks like:

```
(:no-error lambda-list . body)
```

If execution of the form completes normally and there is a `:no-error` clause, the values produced by the form will be bound to variables in the clause's *lambda-list* and the *body* will be executed with none of the handler bindings in effect. In this case the value of the `xcl:condition-case` form is the value returned by the last form of the *body* of its `:no-error` clause. Having a `:no-error` clause is equivalent to wrapping `(multiple-value-call #'(lambda (lambda-list) . body) ...)` around the `xcl:condition-case` form.

`conditions:handler-case` is synonymous with `xcl:condition-case`.

Examples:

```
(xcl:condition-case (/ x y)
  (division-by-zero () nil))

(xcl:condition-case (open *the-file*
                          :direction :input)
  (file-error (condition)
    (format t "~&Open failed: ~A~%" condition)))

(xcl:condition-case (some-user-function)
  (file-error (condition) condition)
  (division-by-zero () 0)
  ((or unbound-variable undefined-function) ()
    'unbound))

(xcl:condition-case (open my-file)
  (file-error ()
    (format *error-output* "Couldn't open ~S."
      my-file))
  (:no-error (stream)
    (open-more-files my-file stream) stream)))
```

Note the difference between `xcl:condition-case` and `conditions:handler-bind`. In `conditions:handler-bind`, you are specifying functions that will be called in the dynamic context of the condition signalling form. In `xcl:condition-case`, you are specifying continuations to be used instead of the original form if a condition of a particular type is signaled. These continuations will be executed in the same dynamic context as the original form.

`conditions:ignore-errors` & *body forms*

[Macro]

Executes the forms in a context that handles conditions of type `error` by returning control to this form. If no error is signaled, all

values returned by the last form are returned by conditions:ignore-errors. Otherwise, the form returns the two values nil and the condition that was signaled. Synonym for

```
(xcl:condition-case (progn . forms)
  (error (condition))
  (values nil condition)).
```

~~xcl:debug &optional datum &rest arguments~~ [Function]

~~Enters the debugger with a given condition without signalling that condition. When the debugger is entered, it will announce the condition by invoking the condition's report function.~~

~~datum is treated the same as for xcl:signal except if datum is not specified, it defaults to "Call to DEBUG".~~

~~This function will never directly return to its caller. Return can occur only by a special transfer of control, such as to a catch, block, tagbody, xcl:proceed-case or xcl:catch-abort.~~

conditions:invoke-debugger condition [Function]

Invokes the debugger with the given condition. This is intended to be used as a portable entry point to the debugger. For finer control over the debugging state, see the function xcl:debugger.

break &optional format-string &rest format-arguments [Function]

Enters the debugger with a simple condition with the given arguments. If no *format-string* is provided, it defaults to "Break." Computation may be continued by invoking the conditions:continue restart. If continued, break returns nil.

break is approximately:

```
(defun break (&optional (format-string "Break")
              &rest format-arguments)
  (conditions:restart-case (conditions:invoke-debugger
    (conditions:make-conditions 'conditions:simple-condition
      :format-string format-string :format-arguments format-arguments)
    (conditions:continue ()
      :report "Return from BREAK."
      nil)))
```

Restarts

conditions:restart-case expression {(case-name arglist {keyword value}* {form}*)}*

[Macro]

The *expression* is evaluated in a dynamic context where the case clauses have special meanings as points to which control may be transferred. If *expression* runs to completion, all values returned by the form are simply returned by the conditions:restart-case form. On the other hand, the computation of *expression* may choose to transfer control to one of the restart clauses. If a transfer to a clause occurs, the forms in the body of that clause will be

evaluated in the same dynamic context as the `conditions:restart-case` form, and any values returned by the last such form will be returned by the `conditions:restart-case` form.

A restart clause has the form given above:

```
(case-name arglist {keyword value}* {form}*)
```

The *case-name* may be `nil` or any symbol.

The *arglist* is a normal lambda list that will be bound and evaluated in the dynamic context of the `conditions:restart-case` form. They will use whatever values were provided by `conditions:invoke-restart` or `conditions:invoke-restart-interactively`. Definitions of these two functions appear later in this section.

The valid *keyword/value* pairs are:

```
:filter expression
```

expression should be suitable as an argument to the function special form. It defines a predicate of no arguments that determines if this clause is visible to `conditions:find-restart`. Default = `true`.

```
:condition type
```

Shorthand for a common special case of `:filter`. The following two *key/value* pairs are equivalent:

```
:condition foo  
:filter  
  (lambda ()  
    (typep xcl:*current-condition*  
      'foo))
```

```
:interactive expression
```

The *expression* must be a form suitable as an argument to function. (`function expression`) will be evaluated in the current lexical and dynamic environments. The result should be a function of no arguments which returns a list of values to be used by `conditions:invoke-restart-interactively`. This function will be called in the dynamic environment available prior to any restart attempt. Any interaction with the user should be done here and not in the body of the restart.

If there is no `:interactive` option specified and the restart is invoked interactively, no arguments will be supplied.

```
:report expression
```

The *expression* can either be a constant string or a form suitable as an argument to function.

If *expression* is not a string, (`function expression`) will be evaluated in the current lexical and dynamic environment. The

result should be a function of one argument, a stream, which will be called to report that restart. This function should print a short summary of the action that restart will take if invoked.

If *expression* is a string, it is a shorthand for `(lambda (s) (format s expression))`.

Only one of `:condition` or `:filter` may be specified. If no `:report` is specified, the *case-name* will be used. It is an error to have a null case name and no report function.

Examples:

```
(loop
  (conditions:restart-case
    (return (apply function some-args))
    (new-function (new-fn)
      :report "Use a different function."
      :interactive (lambda ()
                     (list (prompt-for 'function "Function:
"))))
    (setq function new-fn))))

(loop
  (conditions:restart-case
    (return (apply function some-args))
    (nil (new-fn)
      :report "Use a different function."
      :interactive (lambda ()
                     (list (prompt-for 'function "Function:
"))))
    (setq function new-fn))))

(conditions:restart-case (a-command-loop)
  (return-from-command-level ()
    :report
      (lambda (stream)
        (format stream "Return from command level ~D."
level)))
  nil))

(loop
  (conditions:restart-case (another-computation)
    (conditions:continue () nil)))
```

The first and second examples are equivalent from the point of view of someone using the interactive debugger, but differ in one important aspect for non-interactive handling. If a handler "knows about" restart names, as in:

```
(when (conditions:find-restart 'new-function)
  (conditions:invoke-restart 'new-function the-
replacement))
```

then only the first example, and not the second, will have control transferred to its correction clause.

Here's a more complete example:

```
(let ((my-food 'milk)
      (my-color 'greenish-blue))
  (do ()
```

```
((not (food-colorable-p my-food
                        my-color)))
(conditions:restart-case (error 'bad-food-color
                              :food my-food
                              :color my-color)
  (use-food (new-food)
    :report "Use another food."
    (setf my-food new-food))
  (use-color (new-color)
    :report "Use another color."
    (setf my-color new-color))))
;; We won't get to here until my-food
;; and my-color are compatible.
(list my-food my-color))
```

Assuming that use-food and use-color have been defined as

```
(defun use-food (new-food)
  (invoke-restart 'use-food new-food))

(defun use-color (new-color)
  (invoke-restart 'use-color new-color))
```

then a handler can proceed from the error in either of two ways. It may correct the color or correct the food. For example:

```
#'(lambda (condition) ...
    ;; Corrects color
    (use-color 'white) ...)

or

#'(lambda (condition) ...
    ;; Corrects food
    (use-food 'cheese) ...)
```

Here is an example using conditions:handler-bind and conditions:restart-case.

```
(conditions:handler-bind ((foo-error
                          #'(lambda (condition)
                              (conditions:use-value 7))))
  (conditions:restart-case (error 'foo-error)
    (conditions:use-value (x) (* x x))))
```

The above form returns 49.

```
xcl:define-proceed-function name [Macro]
_____
_____ {keyword value}* _____
_____ {variable}* _____
```

~~Valid keyword/value pairs are the same as those which are defined for the xcl:proceed-case special form. That is, :filter, :filter-function, :condition, :report, and :report-function. The filter and report functions specified in a xcl:define-proceed-function form will be used for xcl:proceed-case clauses with the same name that do not specify their own filter or report functions, respectively.~~

~~This form defines a function called name which will invoke a proceed case with the same name. The proceed function takes optional arguments which are given by the variables specification. The parameter list for the proceed function will look like~~

~~—(&optional . *variables*)~~

The only thing that a *proceed* function really does is collect values to be passed on to a *proceed* case clause.

Each element of *variables* has the form *variable-name* or (*variable-name initial-value*). If *initial-value* is not supplied, it defaults to *nil*.

For example, here are some possible *proceed* functions which might be useful in conjunction with the *bad-food-color* error we used as an example earlier:

```
(xcl:define-proceed-function use-food
  :report "Use another food."
  (food (read-typed-object 'food
    "Food to use instead: ")))

(xcl:define-proceed-function use-color
  :report "Change the food's color."
  (color
    (read-typed-object 'food
      "Color to make the food: ")))

(defun maybe-use-water (condition)
  ;; A sample handler
  (when (eq (bad-food-color food condition)
    'milk)
    (use-food 'water)))

(xcl:handler-bind ((bad-food-color
  #'maybe-use-water))
  ...)
```

If a named *proceed* function is invoked in a context in which there is no active *proceed* case by that name, the *proceed* function simply returns *nil*. So, for example, in each of the following pairs of handlers, the first is equivalent to the second but less efficient:

```
#'(lambda (condition) ; OK, but slow
  (when (xcl:find-proceed-case 'use-food)
    (use-food 'milk)))
#'(lambda (condition) ; Preferred
  (use-food 'milk))

#'(lambda (condition)
  (cond ((xcl:find-proceed-case 'use-food)
    (use-food 'chocolate))
        ((xcl:find-proceed-case 'use-color)
    (use-color 'orange))))
#'(lambda (condition)
  (use-food 'chocolate)
  (use-color 'orange)))
```

conditions:restart-bind (((*name function {keyword value}**)) * { *form* } * [Macro]

Executes the *forms* in a dynamic context where the given restart bindings are in effect.

name may be *nil* to indicate an anonymous restart, or some other symbol to indicate a named restart.

function will be evaluated in the current lexical and dynamic contexts and should produce a function of no arguments to be used to perform the restart. This function will be called when that restart is activated by `conditions:invoke-restart` or `conditions:invoke-restart-interactively`. Note that unlike `conditions:restart-case`, invoking the restart does not automatically transfer control back to the contour in which it was established. If that is appropriate for that restart it is up to the individual restart function to do this.

The valid *keyword/value* pairs are:

`:interactive-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of no arguments that will construct the list of values to be used by `conditions:invoke-restart-interactively`.

`:report-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of one argument, a stream, that will be used to report that restart.

`:filter-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of no arguments that will be used to determine if the given restart is currently active.

This form is a more primitive way of establishing restarts than `conditions:restart-case`. It is expected that `conditions:restart-case` will be sufficient for most uses of the restart facility. An example of where the more general facility provided by `conditions:restart-bind` may be useful is:

```
(conditions:restart-bind ((nil #'(lambda ()
(expunge-directory the-dir)) :report-function
#'(lambda (stream) (format stream "Expunge ~A."
(directory-namestring the-dir)))) (cerror "Try
this file operation again." 'directory-full
:directory the-dir))
```

In this case, a restart is provided that allows the user to expunge the full directory and return to the debugger after doing so. He can then try some other restart, such as `conditions:continue` to retry the failed operation.

`conditions:compute-restarts`**[Function]**

Uses the dynamic state of the program to compute a list of *restarts*.

Each restart object represents a point in the current dynamic state of the program to which control may be transferred. The only operations that Lisp defines for such objects are:

```
conditions:restart-name,
conditions:find-restart,
conditions:invoke-restart, conditions:invoke-
restart-interactively,
princ, and
prinl,
```

to identify an object as a restart using (`typep x 'conditions:restart`), and standard Lisp operations that work for all objects, such as `eq`, `eql`, `describe`, etc.

The list which results from a call to `conditions:compute-restarts` is ordered so that the innermost (ie, more-recently established) restarts are nearer the head of the list.

Note also that `conditions:compute-restarts` returns *all* valid restarts, even if some of them have the same name as others and therefore would not be found by `conditions:find-restart`.

It is an error to modify the list returned by `conditions:compute-restarts`.

conditions:restart-name *restart* [Function]

Returns the name of the given *restart*, or `nil` if it is not named.

~~xcl:default-proceed-test~~ *proceed-case-name* [Macro]

~~Returns the default filter function for proceed cases with the given *proceed-case-name*. May be used with `setf` to change it.~~

~~xcl:default-proceed-report~~ *proceed-case-name* [Macro]

~~Returns the default report function for proceed cases with the given *proceed-case-name*. This may be a string or a function just as for condition types. May be used with `setf` to change it.~~

conditions:find-restart *identifier* [Function]

Searches for a restart by the given *identifier* which is in the current dynamic environment.

If *identifier* is a symbol, then the innermost (ie, most recently established) restart with that name that is active is returned. `nil` is returned if no such restart is found.

If *identifier* is a restart object, then it is simply returned unless it is not currently valid for use. In that case, `nil` is returned.

When searching for a matching restart, the filter function, if any, of potential matches will be called to see if they are active. If it returns

`nil`, then the restart is considered to not have been seen and the search for a match continues.

Although anonymous restarts have a name of `nil`, it is an error for the symbol `nil` to be given as an *identifier* to this function. If it is appropriate to search for anonymous restarts, you should use `conditions:compute-restarts` instead.

`conditions:invoke-restart` *restart* &**rest** *values*

[Function]

Calls the function associated with the given *restart*, passing the *values* as arguments. The *restart* must be a restart object or the non-null name of a restart which is valid in the current dynamic context. If an argument is not valid, an error of type `conditions:control-error` will be signalled.

~~If the argument is a named proceed case that has a corresponding proceed function, `xcl:invoke-proceed-case` will do the optional argument resolution specified by that function before transferring control to the proceed case.~~

`conditions:invoke-restart-interactively` *restart*

[Function]

Calls the function associated with the given *restart*, providing for any necessary arguments. The *restart* must be a restart object or the non-null name of a restart which is valid in the current dynamic context. If the *restart* is not valid, an error of type `conditions:control-error` will be signalled.

`conditions:invoke-restart-interactively` will first call the *restart's* interactive function as specified by the `:interactive` keyword of `conditions:restart-case` or the `:interactive-function` keyword of `conditions:restart-bind`. The interactive function should return a list of values to be passed as arguments to the *restart*. This list must be at least as long as the number of required arguments that the *restart* has.

If the *restart* has no interactive function, no arguments will be passed to the restart function. It is an error for a restart to require arguments but not have an interactive function.

Once the arguments have been determined, `conditions:invoke-restart-interactively` will simply do `(apply #'conditions:invoke-restart restart arguments)`.

`conditions:with-simple-restart` (*name* *format-string* {*format-arguments*}*) {*form*}*

[Macro]

This is a shorthand for one of the most common uses of `conditions:restart-case`.

If the *restart* designated by *name* is not invoked while executing the *forms*, all values produced by the last *form* are returned. If the restart established by `conditions:with-simple-restart` is

invoked, control is transferred to the `conditions:with-simple-restart` form, which immediately returns the two values `nil` and `t`.

It is permissible for *name* to be `nil`. In that case, an anonymous restart is established.

`conditions:with-simple-restart` is essentially:

```
(defmacro conditions:with-simple-restart
  ((restart-name format-string
    &rest format-arguments)
   &body forms)
  `(conditions:restart-case (progn ,@forms)
    (,restart-name ()
     :report (lambda (stream)
                (format stream
                        ,format-string
                        ,@format-arguments)))
    (values nil t))))
```

Example:

```
(defun read-eval-print-loop (level)
  (conditions:with-simple-restart
    (conditions:abort "Exit command level ~D." level)
    (loop
      (conditions:with-simple-restart
        (conditions:abort "Return to command level ~D."
          level)
          (print (eval (read)))))))
```

xcl:catch-abort *print-form* &**body** *forms*

[Macro]

Like `conditions:with-simple-restart`, but always uses the name `conditions:abort`.

`xcl:catch-abort` could be defined by:

```
(defmacro xcl:catch-abort (print-form
                           &body forms)
  `(conditions:with-simple-restart
    (conditions:abort ,print-form)
    ,@forms))
```

conditions:abort

[Function]

This function transfers control to the nearest active restart named `conditions:abort`. If there is none, this function signals an error of type `conditions:control-error`.

~~`xcl:abort` could be defined by:~~

```
(define-procedure-function xcl:abort
  :report "Abort")
```

conditions:continue

[Function]

This function transfers control to the nearest active restart named `conditions:continue`. If none exists it simply returns `nil`.

The `conditions:continue` restart is generally part of simple protocols where there is a single "obvious" way to continue, such as in `break` and `cerror`.

NB: `conditions:continue` replaces `xcl:proceed`.

~~`xcl:proceed` & optional `condition`~~ [Function]

~~This is a predefined `proceed` function. It is used by such functions as `break`, `cerror`, etc.~~

`conditions:muffle-warning` [Function]

This function transfers control to the nearest active restart named `conditions:muffle-warning`. If none exists, an error of type `conditions:control-error` is signalled.

`warn` sets up this restart so that handlers of `conditions:warning` conditions have a way to tell `warn` that the warning has been dealt with and that no further action is warranted.

`conditions:use-value` *new-value* [Function]

This function transfers control (and one value) to the nearest active restart named `conditions:use-value`. If no such restart exists, this function simply returns `nil`.

The `conditions:use-value` restart is generally used by handlers trying to recover from errors of types such as `conditions:cell-error`, where the handler may wish to supply a replacement datum for one-time use.

`conditions:store-value` *new-value* [Function]

This function transfers control (and one value) to the nearest active restart named `conditions:store-value`. If no such restart exists, this function simply returns `nil`.

The `conditions:store-value` restart is generally used by handlers trying to recover from errors of types such as `conditions:cell-error`, where the handler may wish to supply a replacement datum to be stored in the offending cell.

[This page intentionally left blank]

A

Abort (Editor Command) B-7
ACCESS 3-38
Add-Command (Function) B-14
add.process (Function) 4-12; 7-12
ADDMENU (Function) 4-24
ADDTOSCRATCHLIST (Function) 4-1
ADVICE (File Manager Command) 3-15
ADVINFOLST (Variable) 3-14
ADVISE (File Manager Command) 3-15
ADVISE (Function) 3-13,15
ADVISEDVNS (Variable) 3-14
ADVISEDUMP (Function) 3-14
 Advising 3-14; 7-9
AFTERDOMAKESYS 4-7
AFTERDOSAVEVM 4-7
AFTERDOSYSOUT 4-7
AFTERLOGOUT 4-7
AFTERLOGOUTFORMS 4-7
AFTERMAKESYS 4-7
AFTERSAVEVM 4-7
AFTERSYSOUT 4-7
AGAIN (Editor Command) B-8
ALL (Event Address) A-5
ALLOWED-LOGINS 4-6
append (Function)
 with non-list argument 7-8
 Application Menus D-1
APPLY-format input A-3
ARCHIVEFLG (Variable) 3-9
ARCHIVEFN (Variable) 3-9
Arglist (Editor Command) B-9
AROUNDEXITFNS (Variable) 4-7
 array reference 7-4
 arrays 3-3
ASKUSER (Function) 4-16
assert (Macro) E-10
Attach Menu (Editor Command) B-11
 Attached Windows 4-28
AUTHENTICATE 4-6
AUTHENTICATION.NET.HINT (Variable) 4-33
AUTOHARDRESETFLG 4-5

B

back-quote facility 3-49
BACKGROUND (FreeMenu Group Property) D-8
BACKGROUND (FreeMenu Item Property) D-10
BACKGROUNDVNS (Variable) 4-12
BACKSPACE (Editing Command) A-21
BCOMPL (Function) 3-22,25; 4-10
BEEPON (Function) 4-31
BEFORELOGOUT 4-7
BEFOREMAKESYS 4-7
BEFORESAVEVM 4-7
BEFORESYSOUT 4-7
BEFORESYSOUTFORMS 4-7
BITMAP (FreeMenu System Property) D-10
BKSYSBUF (Function) 4-30
BKSYSCHARCODE (Function) 4-30
BLOCKRECORD (Record Type) 4-3
BOTTOM (FreeMenu Group Property) D-7
 bound E-4

BOUNDP (Function) 3-2

BOX (FreeMenu Group Property) D-5,8

BOX (FreeMenu Item Property) D-10

BOXSHADE (FreeMenu Group Property) D-8

BOXSHADE (FreeMenu Item Property) D-10

BOXSPACE (FreeMenu Group Property) D-8

BOXSPACE (FreeMenu Item Property) D-10

break (Function) 3-13; E-13

break commands 3-13

Break packages 3-9

BREAK0 (Function) 3-13

BREAK1 (Function) 3-9

BREAKCONNECTION (Function) 4-14

BREAKIN (Function) 3-13

breaking 7-9

BREAKREGIONSPEC (Variable) 4-8

BRECOMPILE (Function) 3-22,25

BRKINFOLST (Variable) 3-13

BROKENFNS (Variable) 3-13

bulk data transfer 4-34

C

Catch errors 3-10

ccase (Macro) E-10

cerror (Function) E-9

Change Print Base (Editor Command) B-11

CHANGBACKGROUND (Function) 4-31

CHANGEFONT (Function) 4-23

CHANGESLICE (Function) A-11,17

CHANGESTATE (FreeMenu Item Property) D-11

changing a standard readtable 3-22

characters 3-3

CHARCODE (Function) 3-3

CHCON (Function) 3-42

check-type (Macro) E-10

CL Exec 3-7

CL:* (Variable) A-10

CL:** (Variable) A-10

CL:*** (Variable) A-10

CL:+ (Variable) A-10

CL:++ (Variable) A-10

CL:+++ (Variable) A-10

CL:- (Variable) A-10

CL:/ (Variable) A-10

CL:// (Variable) A-10

CL:/// (Variable) A-10

CL:BREAK (Function) 3-13

CL:CATCH (Function) 3-5

CL:CODE-CHAR (Function) 3

CL:COMPILE-FILE (Function) 3-24-25; 4-10

CL:DEFCONSTANT (Variable) 3-20

CL:DEFINE-MODIFY-MACRO (Function) 3-20

CL:DEFMACRO (Function) 3-20

CL:DEFMACRO (Macro) 3-29

CL:DEFPARAMETER (Macro) 3-26,29

CL:DEFPARAMETER (Variable) 3-20

CL:DEFUN (Function) 3-20

CL:DEFUN (Macro) 3-29

CL:DEFVAR (Macro) 3-29

CL:DEFVAR (Variable) 3-20

CL:ERROR 3-10

CL:EVAL-WHEN (File Package Command) 3-31

CL:GENSYM (Function) 3-2
CL:LOAD (Function) 3-24
CL:MAKE-HASH-TABLE (Function) 3-4
CL:MAPHASH (Function) 3-4
CL:PRIN1 (Function) 3-41-42
CL:PRINC (Function) 3-41
CL:READ (Function) 3-40
CL:READ-PRESERVING-WHITESPACE (Function) 3-41
CL:THROW (Function) 3-5,11
CL:UNWIND-PROTECT 3-6
CL:UNWIND-PROTECT (Function) 3-11
CL:WITH-INPUT-FROM-STRING 3-37
CL:WRITE (Function) 3-41
CLEANUP (Function) 3-25
cleanup forms 3-6
CLEARCLISPARRAY (Function) 4-10
CLEARSTK (Function) 4-5
CLEARSTKLST (Variable) 4-5
CLISP infix forms 3-33
CLISPARRAY 4-2
CLOSEALL (Function) 3-38
closure 7-8
coerce (Function) 7-12
COERCE-TO-NSADDRESS (Function) 4-33
collect (Macro) 7-6
collecting objects
 macros for 7-6
COLLECTION (FreeMenu Item Property) D-12
COLLECTION property 4-26
COLUMN (FreeMenu Group Property) D-7
COLUMNSPACE (FreeMenu Group Property) D-7
Comment Out Selection (Editor Command) B-9
comment treated as declaration 3-32
Comments
 in SEdit B-6
Common Lisp strings 3-3
Common Lisp Symbols 3-1
COMMONNUMSYNTAX 3-44
compile-definer (Definer) 7-2
compile-form (Definer) 7-2
compiler
 behavior with FLETed lexical functions 7-12
 behavior with recursion 7-12
 ignoring TEdit formatting 7-12
 retaining special arguments 7-12
complex numbers 3-4
coms 7-11
condition E-3
conditions:*break-on-signals* (Variable) E-9
conditions:abort (Function) E-21
conditions:compute-restarts (Function) E-18
conditions:continue (Function) E-21
conditions:define-condition (Macro) E-5
conditions:find-restart (Function) E-19
conditions:handler-bind (Macro) E-4,11
conditions:handler-case (Macro) E-11
conditions:ignore-errors (Macro) E-12
conditions:invoke-debugger (Function) E-13
conditions:invoke-restart (Function) E-5,20
conditions:invoke-restart-interactively (Function) E-20
conditions:make-condition (Function) E-6,8
conditions:muffle-warning (Function) E-22
conditions:restart-bind (Macro) E-17
conditions:restart-case (Function) E-5
conditions:restart-case (Macro) E-13
conditions:restart-name (Function) E-19

conditions:signal (Function) E-8
conditions:store-value (Function) E-22
conditions:use-value (Function) E-22
conditions:with-simple-restart (Macro) E-20
CONN (Exec Command) A-7
CONTROL-A (Editing Command) A-21
Control-C (Editor Command) B-7
Control-L (Editor Command) B-7
Control-Meta-; (Editor Command) B-9
Control-Meta-F (Editor Command) B-8
Control-Meta-O (Editor Command) B-7
Control-P 4-29
CONTROL-Q (Editing Command) A-21
CONTROL-R (Editing Command) A-21
Control-T 4-29
CONTROL-W (Editing Command) A-21
Control-W (Editor Command) B-7
CONTROL-X (Editing Command) A-21
Control-X (Editor Command) B-7
Convert Comments (Editor Command) B-9
Convert-Upgrade (Variable) B-14
converting characters 3-3
Converting old code
 for use with new Error system E-1
COORDINATES (FreeMenu Group Property) D-7
COPY (Function) 3-49
COPYBYTES (Function) 4-16
COPYDEF (Function) 4-4
COPYFILE (Function) 3-38
COPYREADTABLE (Function) 3-46
COS (Function) 4-3
COURIER.CALL (Function) 4-34
COURIER.OPEN (Function) 4-34
Creating an Exec process A-18
Creating conditions E-4
Creating icons
 with ICONW C-1
CTRLUFLG 4-18
ctypecase (Macro) E-10
CUHOTSPOTX 4-30
CUHOTSPOTY 4-30
CUIMAGE 4-30
current package 3-45
CURSOR 4-30
Cursor Movement Commands A-22
CURSORBITMAP 4-30
CURSORCREATE (Function) 4-30
CURSORHOTSPOTX 4-30
CURSORHOTSPOTY 4-30

D

DA (Exec Command) A-7
DAUGHTERS (FreeMenu Group Property) D-8
DC (Function) 3-18
Declining by Condition handler E-4
DEdit 3-15
Default handlers 3-10
Default-Commands (Function) B-15
DEFAULT.OSTYPE (Variable) 4-15
DEFAULTFONT (Variable) D-7
DEFAULTICONFN (Variable) 4-25
DEFAULTTEXTICON (Variable) C-3
deferredconstant (Function) 7-12
define-file-environment (Definer) 7-2
define-record (Definer) 7-3
Defining New Terms A-11
DEFMACRO (Macro) 3-5
defstruct (Macro) 7-4

warning 7-6
DELDEF (Function) 3-28
Delete Selection (Editor Command) B-7
Delete Structure (Editor Command) B-8
Delete Word (Editor Command) B-7
DELFILE (Function) 3-38
DESELECT (FreeMenu Item Property) D-12
DF (Function) 3-18
DFASL files 2-1
DFNFLG (Variable) 3-27
DIR (Exec Command) A-7
DISPLAY (FreeMenu Item) D-6-7,14
Display icons C-1
DISPLAY item 4-26
DISPLAYFONTDIRECTORIES (Variable) 4-23
DMACRO (Property) 3-5
DMACROs 2-1
DO-EVENTS (Exec Command) A-8
DOCOLLECT (Function) 4-1
DOSHAPFN (Window Property) 4-25
DOWNFN (FreeMenu Mouse Property) D-10
DP (Function) 3-18
DRAWARC (Function) 4-19
DRAWLINE (Function) 4-19
DRAWPOLYGON (Function) 4-20
DSPCLEOL (Function) 4-18
DSPFONT 4-16
DSPRUBOUTCHAR (Function) 4-18
DPSSCALE 4-19
dummy definitions 3-17
DV (Function) 3-18
DWIMIFYCOMPFLG (Variable) 3-34

E

ecase (Macro) E-10
ECHOCHAR (FreeMenu Item Property) D-13
ED (Function) 3-16
Edit (Editor Command) B-9
EDIT (FreeMenu Item) 4-27; D-13
Edit caret in SEdit B-2
Edit Interface 3-18
EDITBM (Function) 4-18
EDITCALLERS (Function) 3-19
Editing Exec Input A-20
Editing Lisp Code in Memory B-1
Editing VALUES 3-18
EDITMODE (Function) 3-16
EDITSTART (FreeMenu Item) 4-27; D-14
END-OF-FILE (Error Type) 3-12
ENDCOLLECT (Function) 4-1
Ending an SEdit session B-2
ENDOFSTREAMOP 3-38
ENVAPPLY 3-6
ENVEVAL 3-6
EQUAL (Function) 3-26
EQUALALL (Function) 4-3
ERROR (Function) 3-10
error (Function) E-9
Error conditions 3-10
error system 3-10
Error system
differences between old and new E-1
Error system proposal E-1
Error type mapping 3-11
Error type name 3-11
Error type number 3-11
ERROR! (Function) 3-10
ERRORMESS (Function) 3-10

ERRORMESS1 (Function) 3-10

ERRORN (Function) 2-2; 3-10

Errors

definition of E-3

ERRORSET 3-10

ERRORSTRING (Function) 3-10

ERRORTYPELIST 3-10

ERRORTYPELIST (Variable) 2-2

ERSETQ (Function) 3-10; 4-8

ERXM 3-10

ESCAPE (Editing Command) A-21

Escape

in SEdit B-6

Establishing handlers within dynamic context E-4

etypcase (Macro) E-10

Eval (Editor Command) B-9

EVAL-format input A-2

Exec Editing Commands A-22

Exec type A-4

EXEC-EVAL (Function) 3-9

EXPAND (Editor Command) B-9

EXPANDBITMAP (Function) 4-18

EXPANDMACRO (Function) 3-5

EXPANDREGIONFN (Window Property) 4-24

EXPLICIT (FreeMenu Group Property) D-7

export (Function) 7-9

Extract (Editor Command) B-9

F

F (Event Address) A-5

features

new Common Lisp 7-1

FETCH 3-33

File Manager 3-19

file-reading functions 3-20

FILEPKGCOM (Function) 4-9

FILEPKGTYPE (Function) 4-9

FILEPKGYPES (Variable) 3-16

FILEPOS (Function) 4-16

FILERDTBL 3-22

files containing bitmaps 3-31

FILES? (Function) 3-28

FILETYPE (Property) 3-25

FILLPOLYGON (Function) 4-19-20

FIND (Editor Command) B-8

Find Gap (Editor Command) B-8

FIND-READTABLE (Function) 3-45

FINDCALLERS (Function) 3-19

FIX (Exec Command) A-8

FIXP (Predicate) 3-4

flet (Special form) 7-4

floating point 3-4

FLOATP (Predicate) 3-4

FM.BACKGROUND (FreeMenu Window Property)
D-15

FM.CHANGELABEL (FreeMenu Function) D-16

FM.CHANGELABEL (Function) 4-27-28

FM.CHANGESTATE (FreeMenu Function) D-16

FM.CHANGESTATE (Function) 4-28

FM.DONTRESHAPE (FreeMenu Window Property)
D-15

FM.EDITITEM (FreeMenu Function) D-17

FM.EDITP (FreeMenu Function) D-17

FM.ENEDIT (FreeMenu Function) D-17

FM.FIXSHAPE (Function) 4-28

FM.FORMATMENU (Function) 4-26-27

FM.GETITEM (Function) 4-27

FM.GETITEM (FreeMenu Function) D-15

FM.GETSTATE (*FreeMenu Function*) D-16
FM.GETSTATE (*Function*) 4-27
FM.GROUPPROP (*FreeMenu Macro*) D-7,18
FM.HIGHLIGHTITEM (*FreeMenu Function*) D-17
FM.HIGHLIGHTITEM (*Function*) 4-28
FM.ITEMFROMID (*Function*) 4-27
FM.ITEMPROP (*FreeMenu Macro*) D-18
FM.MAKEMENU (*Function*) 4-26-27
FM.MENUPROP (*FreeMenu Macro*) D-7,19
FM.NWAYPROP (*FreeMenu Macro*) D-19
FM.NWAYPROPS (*Macro*) 4-27
FM.PROMPTWINDOW (*FreeMenu Window Property*) D-15
FM.READSTATE (*Function*) 4-27
FM.REDISPLAYITEM (*FreeMenu Function*) D-18
FM.REDISPLAYMENU (*FreeMenu Function*) D-18
FM.RESETGROUPS (*FreeMenu Function*) D-17
FM.RESETMENU (*FreeMenu Function*) D-17
FM.RESETSHAPE (*FreeMenu Function*) D-17
FM.RESETSHAPE (*Function*) 4-28
FM.RESETSTATE (*FreeMenu Function*) D-17
FM.SHADE (*FreeMenu Function*) D-18
FM.SHADE (*Function*) 4-28
FM.SHADEITEM (*Function*) 4-28
FM.SHADEITEMBM (*Function*) 4-28
FM.SKIPNEXT (*FreeMenu Function*) D-17
FM.TOPGROUPID (*FreeMenu Function*) D-18
FM.WHICHITEM (*FreeMenu Function*) D-18
FONT (*FreeMenu Group Property*) D-7
FONT (*FreeMenu Item Property*) D-9
font descriptor 4-22
FONTCHANGEFLG (*Variable*) 4-23
FONTCREATE (*Function*) 4-22
FONTSAVAILABLE 4-21
FOR 3-33
FOR (*Exec Command*) A-6
FORGET 4-6
FORGET (*Exec Command*) A-8
FORMAT (*FreeMenu Group Property*) D-4,7
Free Menu
 How to make a D-1
Free Menu format D-2
Free Menu layout D-1
FREEMENU (*FreeMenu Function*) D-15
FREEMENU (*Function*) 4-26-27
FROM (*Event Address*) A-5
FULLNAME (*Function*) 3-37
FUNARG 4-4

G

Gaps

 in SEdit B-4

garbage collector 4-11

gensym (*Function*) 3-2; 7-12

GET-ENVIRONMENT-AND-FILEMAP (*Function*) 3-23

Get-Prompt-Window (*Function*) B-15

Get-Selection (*Function*) B-16

Get-Window-Region (*Function*) B-13

GETDEF (*Function*) 3-28

GETFILEINFO (*Function*) 3-38; 4-13

GETPROMPTWINDOW (*Function*) 4-28

GETREADTABLE (*Function*) 3-39

GETSYNTAX 3-45

global macro shadowing 7-4

GROUP (*FreeMenu Group Property*) D-7

GROUPID (*FreeMenu System Property*) D-10

H

handler (*Function*) E-4

Handling conditions E-3

HARDCOPYW (*Function*) 4-29

HARDRESET (*Function*) 4-4

HASDEF (*Function*) 3-26,28; 4-9

hash arrays 3-4

HASHARRAY 3-4

HASHARRAY (*Function*) 4-2

HELDFN (*FreeMenu Mouse Property*) D-10

HELP (*Editor Command*) B-9

HELP (*Function*) 3-10

Help Menu Commands B-11

HIGHLIGHT (*FreeMenu Item Property*) D-9,14

History list A-16

HISTORYSAVEFORMS (*Variable*) 3-9

HJUSTIFY (*FreeMenu Item Property*) D-4,9

HORRIBLEVARS 4-9,15

HPRINT (*Function*) 4-15

I

ICONW (*Function*) C-1

ICONW windows

 from an image defined by a mask C-1

 with titles C-1

ICONW.SHADE (*Function*) C-2

ICONW.TITLE (*Function*) C-2

ID (*FreeMenu Group Property*) D-7

ID (*FreeMenu Item Property*) D-9

IDLE-PROFILE 4-6

IDLE-RESETVARS (*Variable*) 4-6

IDLE-SUSPEND-PROCESS.NAMES (*Variable*) 4-7

IEEE 802-3 specification 4-34

IF 3-33

IL Exec 3-7

IL:IT (*Variable*) A-9

IL:LOAD (*Function*) 3-24

IL:MAPHASH (*Function*) 3-4

IL:PRIN1 (*Function*) 3-41

IL:PRIN2 (*Function*) 3-41

IL:READ (*Function*) 3-40

ILLEGAL-GO (*Error Type*) 3-11

ILLEGAL-RETURN (*Error Type*) 3-11

ILLEGAL-STACK-ARG (*Error Type*) 3-12

IN (*Exec Command*) A-6

in-package (*Function*) 7-8

INFILEP (*Function*) 3-37

INFINITEWIDTH (*FreeMenu Item Property*) D-13

INITSTATE (*FreeMenu Item Prop*) 4-26

INITSTATE (*FreeMenu Item Property*) D-9,12

INPUT (*Function*) 3-37

INPUTFONT (*Variable*) A-10

Inspect (*Editor Command*) B-10

INTEGERLENGTH (*Function*) 4-3

integers 3-4

Interlisp Compiler 3-31

INTERLISP-ERROR (*Error Type*) 3-12

INTERPRESSFONTDIRECTORIES (*Variable*) 4-22

INTERRUPTCHAR (*Function*) 4-29

INVALID-ARGUMENT-LIST (*Error Type*) 3-12

ITEMS (*FreeMenu Group Property*) D-8

J

Join (*Editor Command*) B-10

K

Keep-Window-Region (*Variable*) B-13

KEYACTION (Function) 4-31**KEYDOWNP** (Function) 4-31**L****LABEL** (FreeMenu Item Property) D-9

LABELS construct

warning 7-10

LASTC (Function) 4-15

Layout

of Free Menu D-1

LCOM files 2-1

ldflg 7-11

LEFT (FreeMenu Group Property) D-7**LEFT and BOTTOM** (FreeMenu Item Property) D-9

Left mouse button

in SEdit B-3

lexical bindings 3-33

Library modules

summary of changes 5-1

LIMITCHARS (FreeMenu Item Property) D-3,13**LINKS** (FreeMenu Item Property) D-10,15

LISP 3-47

Lisp structures

SEdit gaps for B-4

LISPSOURCEFILEP (Function) 4-10**LISPXEVAL** (Function) 3-9**LISPXFNS** (Variable) A-15**LISPXHISTORY** (Variable) A-16**LISPXHISTORYMACROS** (Variable) 3-9**LISPYMACROS** 3-8**LISPYMACROS** (Variable) 3-9**LISPYREADFN** (Function) 4-8**LISPYUNREAD** (Function) 3-9**LISPYUSERFN** (Variable) 3-9**LIST** (Function) 3-49

Lists

in SEdit B-5

LOAD (Function) 3-20**loadflg** (Argument) 7-11

load-time expression 7-4

LOADCOMP (Function) 3-25**LOADFNS** (Function) 3-20,25**LOADFROM** (Function) 3-25

loading compiled files 3-32

loading Medley files into Lyric 4-10

LOADVARS (Function) 3-25

Locally defined handler E-4

LOCALVARS 3-2**LOGIN.TIMEOUT** 4-6**LOGOUT** (Function) 4-7**long-site-name** (Variable) 7-3**M****MACHINETYPE** (Function) 4-7**MAKE-READER-ENVIRONMENT** (Function) 3-23**MAKEFILE** (Function) 3-20,25,43,49**MAKEFILE-ENVIRONMENT** (Property) 3-21**MAKESYS** (Function) 4-7**MAKETITLEBARICON** 4-25**map** (Function) 7-11**MAPATOMS** (Function) 3-2-3**MAX** (Function) 4-2**MAX.INTEGER** (Variable) 4-2**MAXHEIGHT** (FreeMenu Item Property) D-9**MAXREGION** (FreeMenu System Property) D-11**MAXWIDTH** (FreeMenu Item Property) D-7,9,13

Medley

on Sun workstations 1-1

on Xerox workstations 1-1

Medley compiled files 2-1

Medley enhancements

summary 1-1

MENU (FreeMenu Group Property) D-7**MENUFONT** (FreeMenu Item Property) D-12**MENUIITEMS** (FreeMenu Item Property) D-6,12**MENUTITLE** (FreeMenu Item Property) D-12**MESSAGE** (FreeMenu Item Property) D-9**Meta-** (Editor Command) B-10**Meta-)** (Editor Command) B-10**Meta-/** (Editor Command) B-9**Meta-9** (Editor Command) B-10**Meta-;** (Editor Command) B-9**Meta-A** (Editor Command) B-7**Meta-B** (Editor Command) B-11**Meta-Control-C** (Editor Command) B-7**Meta-Control-S** (Editor Command) B-8**Meta-Control-X** (Editor Command) B-7**Meta-E** (Editor Command) B-9**Meta-F** (Editor Command) B-8**Meta-H** (Editor Command) B-9**Meta-I** (Editor Command) B-10**Meta-J** (Editor Command) B-10**Meta-M** (Editor Command) B-11**Meta-N** (Editor Command) B-8**Meta-O** (Editor Command) B-9**Meta-P** (Editor Command) B-11**Meta-R** (Editor Command) B-8**Meta-Return** (Editor Command) B-10**Meta-S** (Editor Command) B-8**Meta-Space** (Editor Command) B-10**Meta-U** (Editor Command) B-7**Meta-X** (Editor Command) B-9**Meta-Z** (Editor Command) B-10

Middle mouse button

in SEdit B-3

MIN (Function) 4-2**MIN.INTEGER** (Variable) 4-2

minimum window size 4-24

MKSTRING (Function) 3-42**MOMENTARY** (FreeMenu Item) D-11**MOTHER** (FreeMenu Group Property) D-8

Mouse buttons

in SEdit B-3

MOVD (Function) 4-4**MOVEDFN** (FreeMenu Mouse Property) D-10

multiple escape character 3-42

Multiple Execs A-4

multiple streams 3-37

MULTIPLE-ESCAPE 3-45**Mutate** (Editor Command) B-10**N****NAME** (Exec Command) A-8**NCHARS** (Function) 3-42

NCHOOSE item 4-26

NDIR (Exec Command) A-8

Nesting Free Menu Groups D-2

NETWORKOSTYPES (Variable) 4-15**NEW** (MAKEFILE Option) 3-21**NLAMBDA** 3-5**NLSETQ** (Function) 3-10; 4-8**NOBIND** 3-2**NOCLEARSTKLST** (Variable) 4-5**NODIRCORE** (File Device) 4-13**Normalize Selection** (Editor Command) B-10

notational conventions 18
NSADDRESS 4-32
NSNAME 4-32
NSNET.DISTANCE (Function) 4-35
NUMBER (FreeMenu Item) D-14
NUMBERP (Predicate) 3-4
NUMBERTYPE (FreeMenu Item Property) D-14
NWAY (FreeMenu Item) 4-26; D-6; 12
NWAYPROPS (FreeMenu Item Prop) 4-27
NWAYPROPS (FreeMenu Item Property) D-6,12

O

OLD-INTERLISP-FILE 3-47
OLD-INTERLISP-T 3-48
once-only (Macro) 7-7
OPENFILE (Function) 3-37
OPENFN (Window Property) 4-25
OPENP (Function) 3-38
OPENSTREAM (Function) 3-11,37
OPENSTRINGSTREAM (Function) 3-37; 4-16
options E-5
ORIG 3-46
OUTPUT (Function) 3-37

P

package delimiter 2-2
PACKAGEDELIM 3-47
packages 3-19
PARSE-NSADDRESS (Function) 4-33
PAT (Event Address) A-5
pattern matching 3-6
PEEKC (Function) 4-15
pkg-goto (Function) 7-8
PL (Exec Command) A-8
PLVLFILEFLG 3-42
PP (Exec Command) A-9
PRETTYDEF (Function) 4-9
PRIN1 4-30
PRIN2 4-30
PRINT (Function) 3-20,48
PRINTLEVEL 4-29
PRINTNUM (Function) 4-15
PRINTOUT 3-43
PRINTOUTFONT (Variable) A-11
PRINTSERVICE (Variable) 4-19
process status window 4-12
PROCESS.APPLY (Function) 4-12
PROCESS.EVAL (Function) 4-12
Programmer's interface
to SEdit B-12
PROMPT#FLG (Variable) 3-9
PROMPTFONT (Variable) A-10
PROMPTCHARFORMS (Variable) 3-9
PROTECTION 4-13
PRXFLG 3-42
PUTDEF (Function) 3-28

Q

Quote (Editor Command) B-10
Quoted structures
in SEdit B-5

R

RADIX (Function) 3-44
ratios 3-4
READ (Function) 3-20,48
read-eval-print A-1

read/print consistency 3-44
READBUF (Variable) 3-9
READC (Function) 3-41
READER 4-13
READER-ENVIRONMENT 3-20
READLINE (Function) 4-8
READMACROS 4-16
READSYS (Function) 4-35
READTABLEPROP (Function) 3-45
READWISE (Function) 3-14
REALFRAMEP (Function) 4-5
REBREAK (Function) 3-14
RECOMPILE (Function) 3-22,25
record-create (Macro) 7-4
record-fetch (Macro) 7-4
record-ffetch (Macro) 7-4
Redisplay (Editor Command) B-7
Redo (Editor Command) B-8
REDO (Exec Command) A-6
REGION (FreeMenu Group Property) D-8
REGION (FreeMenu System Property) D-11
RELDRAWTO (Function) 4-19
Release Notes
organization of 17
REMEMBER (Exec Command) A-8
REMPROP (Function) 3-2
RENAMEFILE (Function) 3-38
REPAINTFN 4-24
REPAINTFN (Window Property) 4-25
REPEATUNTIL 4-3
Replace-Selection (Function) B-16
Reporting a condition or restart E-5
Reset (Function) 3-10; B-14
Reset-Commands (Function) B-15
RESETFORM 3-40
RESETFORM 3-39
RESETFORMS (Variable) 3-9
RESETLST 3-6
Resetting system state 3-11
RESETVARS 4-6
RESHAPEFN 4-24
Restart type E-5
Restarting computations E-3
Restarting conditions E-5
RETAPPLY 3-6
RETEVAL 3-6
RETFROM 3-6
RETFROM (Function) 3-11
RETRY (Exec Command) A-6
RETTO 3-6
RETURN 3-13; 4-5
Reverse Find (Editor Command) B-8
Right mouse button
in SEdit B-3
ROTATE-BITMAP (Function) 4-18
ROW (FreeMenu Group Property) D-7
row-major-aref (Function) 7-4
ROWSPACE (FreeMenu Group Property) D-7
RS232 or TTY ports 3-38

S

Save-Window-Region (Function) B-13
SAVEVM (Function) 4-7
SCRATCHLIST 4-1
SEdit 3-15
SEdit (Function) B-16
SEdit Command Menu B-12
SEE (Exec Command) A-9

- SEE*** (Exec Command) A-9
SELECTEDFN (FreeMenu Mouse Property) D-10
Set Package (Editor Command) B-11
SETERRORN (Function) 3-10
SETFILEINFO (Function) 3-38; 4-13
SETREADTABLE (Function) 3-48
SETSTKARGNAME (Function) 4-5
SETSYNTAX 3-45,49
SHAPEW (Function) 4-24
SHH (Exec Command) A-8
SHIFT-FIND (Editor Command) B-8
short-site-name (Variable) 7-3
SHOULDCOMPILEMACROATOMS (Variable) 4-4
SHOULDNT (Function) 3-10
SHOWPARENFLG (Variable) A-25
SHRINKBITMAP (Function) 4-18
SHRINKFN (Window Property) 4-24
 SIDE effects of event A-18
 Signalling conditions E-3
SIN (Function) 4-3
 Sketch
 summary of changes 6-10
SKIP-NEXT (Editor Command) B-8
SKREAD (Function) 3-41
SORT (Function) 4-1
 Special characters
 in SEdit B-5
 Specifying event addresses A-4
 Specifying Free Menu Items D-2
 stack manipulations 3-5
STACK OVERFLOW (Error Type) 4-4
 Stack pointers 3-5
STACK-OVERFLOW (Error Type) 3-11
STACK-POINTER-RELEASED (Error Type) 3-12
 Starting an SEdit session B-2
STATE 4-26
STATE (FreeMenu Item) D-7,11
STATE (FreeMenu Item Property) D-12
STATE (FreeMenu System Property) D-10
STKARG (Function) 4-5
STKNARGS (Function) 4-5
STKPOS (Function) 4-5
STOP (Function) 4-10
STOP-UNDOABLY (Macro) A-13
 strings 3-3
 in SEdit B-6
STRINGWIDTH (Function) 3-42; 4-22
 Structure caret in SEdit B-2
 Structure editor 3-15
Substitute (Editor Command) B-8
SUCHTHAT (Event Address) A-5
SUSPEND-PROCESS.NAMES 4-7
 Switching between editors 3-16
 Symbols 3-1,6
 in Error system E-1
 symbols in the INTERLISP package 3-20
SYSDOWNFN (FreeMenu System Property) D-11
 sysload 3-24; 7-11
SYSMOVEDFN (FreeMenu System Property) D-11
YSOUT (Function) 4-7
SYSPRETTYFLG (Variable) 3-9
SYSEXEC (FreeMenu System Property) D-11
- T**
TABLE (FreeMenu Group Property) D-7
TCOMPL (Function) 3-22,25; 4-10
 TEdit
 summary of changes 6-1
 TeleRaid Library module 4-35
TEXTICON (Function) 4-25; C-3
TIME (Exec Command) A-9
TIME (Function) 3-36
TIME (Macro) 3-36
TITLE (FreeMenu Item) 4-27
 titled icons 4-25
TITLEDICONW (Function) C-1
TOGGLE (FreeMenu Item) D-11
TOO-MANY-ARGUMENTS (Error Type) 3-12
TRACE (Function) 3-13-14
TTYBACKGROUNDFN (Variable) 4-12
TTYDISPLAYSTREAM (Function) 4-25
 TTYIN display typein editor 4-16
 TTYIN Editor from Exec A-20
TY (Exec Command) A-9
TYPE (Exec Command) A-9
TYPE (FreeMenu Item Property) D-9
- U**
UGLYVARS 3-43; 4-9,15
UNBOUND-VARIABLE (Error Type) 3-12
UNBREAK (Function) 3-14
UNBREAKIN (Function) 3-13
UNDEFINED-CAR-OF-FORM (Error Type) 3-12
UNDEFINED-FUNCTION-IN-APPLY (Error Type) 3-12
UNDO (Editor Command) B-7
UNDO (Exec Command) A-4,8,13
UNDO key (Editing Command) A-21
UNDOABLY-MAKUNBOUND (Function) 3-29
UNDOABLY-SETQ (Function) A-15
 Undoing in Functions A-14
 Undoing In Programs A-13
 Undoing out of order A-16
UNDOSAVE (Function) A-15
UNIXFTPFLG (Variable) 4-14
UNPACKFILENAME (Function) 3-37
UNSAFEMACROATOMS (Variable) 4-4
UNTIL 4-3
USE (Exec Command) A-6
USERDATA (FreeMenu System Property) D-11
 USERDATA LIST D-14
USEREXEC (Function) 3-9
 USERNAME 4-6
USERWORDS (Variable) A-25
USESILPACKAGE 3-45
 Using Execs 3-7
- V**
VALUEFONT (Variable) A-11
VARS 4-15
 version delimiter 2-2
VIDEORATE (Function) 4-31
VJUSTIFY (FreeMenu Item Property) D-9
- W**
warn (Function) E-10
WHENCHANGED 4-9
WINDOWPROP (Function) 4-26
WINDOWPROPS 4-26
with-collection (Macro) 7-6
with-input-from-string (Macro) 7-13
with-output-to-string (Macro) 7-13
WITH-READER-ENVIRONMENT (Macro) 3-23
write-string (Function) 7-12

WRITESTRIKEFONTFILE (Function) 4-22

writing macros

macros for 7-7

Writing your own SEdit commands B-14

X

XCL 3-47

XCL Compiler 3-31

XCL Exec 3-7

XCL readtable 3-21

xcl:*current-condition* (Variable) E-8**XCL:*DEBUGGER-PROMPT*** (Variable) A-19**XCL:*EVAL-FUNCTION*** (Variable) A-19**XCL:*EXEC-PROMPT*** (Variable) A-19**XCL:*PER-EXEC-VARIABLES*** (Variable) A-18**XCL:ABORT** (Function) 3-10**XCL:ADD-EXEC** (Function) A-18**XCL:ARGLIST** (Variable) 3-15**XCL:ARRAY-SPACE-FULL** (Error Type) 3-12**XCL:ATTEMPT-TO-CHANGE-CONSTANT** (Error Type) 3-11-12**XCL:ATTEMPT-TO-RPLAC-NIL** (Error Type) 3-11**XCL:CATCH-ABORT** 3-10**xcl:catch-abort** (Macro) E-21**XCL:CONDITION** 3-10**xcl:condition-case** (Macro) E-11**xcl:condition-handler** (Macro) E-8**xcl:condition-reporter** (Macro) E-7**XCL:CONTROL-E-INTERRUPT** (Error Type) 3-12**XCL:DATA-TYPES-EXHAUSTED** (Error Type) 3-12**XCL:DEF-DEFINE-TYPE** (Macro) 3-27-28**XCL:DEFCOMMAND** 3-8**XCL:DEFCOMMAND** (Macro) A-11**XCL:DEFDEFINER** (Function) 3-20**XCL:DEFDEFINER** (Macro) 3-29**XCL:DEFGLOBALPARAMETER** (Variable) 3-20**XCL:DEFGLOBALVAR** (Variable) 3-20**XCL:DEFINE-PROCEED-FUNCTION** (Function) 3-20**XCL:DEFININE** (Function) 3-20**XCL:DEFOPTIMIZER** 3-32**XCL:DEFOPTIMIZER** (Macro) 3-5**XCL:EXEC** (Function) A-18**XCL:EXEC-EVAL** (Function) A-19**XCL:EXEC-FORMAT** (Function) A-19**XCL:FILE-NOT-FOUND** (Error Type) 3-12**XCL:FILE-WONT-OPEN** (Error Type) 3-11**XCL:FLOATING-OVERFLOW** (Error Type) 3-12**XCL:FLOATING-UNDERFLOW** (Error Type) 3-12**XCL:FS-PROTECTION-VIOLATION** (Error Type) 3-12**XCL:FS-RESOURCES-EXCEEDED** (Error Type) 3-12**XCL:HASH-TABLE-FULL** (Error Type) 3-12**XCL:INVALID-PATHNAME** (Error Type) 3-12**XCL:SET-DEFAULT-EXEC-TYPE** (Function) A-20**XCL:SET-EXEC-TYPE** (Function) A-20**XCL:SIMPLE-DEVICE-ERROR** (Error Type) 3-11**XCL:SIMPLE-TYPE-ERROR** (Error Type) 3-11**XCL:STORAGE-EXHAUSTED** (Error Type) 3-12**XCL:STREAM-NOT-OPEN** (Error Type) 3-11**XCL:SYMBOL-HT-FULL** (Error Type) 3-11**XCL:SYMBOL-NAME-TOO-LONG** (Error Type) 3-11**XCL:UNDOABLY** (Macro) A-13**XCL:UNDOABLY-SETF** (Macro) A-15**1**

10MB Ethernet encapsulation types 4-34

1108 User's Guide

summary of changes 6-14

1186 User's Guide

summary of changes 6-16

3**3STATE** (FreeMenu Item) 4-26; D-11******\#UNDOSAVES** (Variable) A-15**\10MBTYPE-3TO10** (Variable) 4-34**\10MBTYPE-PUP** (Variable) 4-34**~****~C** (Format directive) 7-13**!****!EVAL** 2-2********break-on-warnings*** (Variable) E-10***Clear-Linear-On-Completion*** (Variable) B-14***Compile-Fn*** (Variable) B-16***COMPILED-EXTENSIONS*** (Variable) 3-25***DEFAULT-CLEANUP-COMPILER*** (Variable) 3-25***DEFAULT-MAKEFILE-ENVIRONMENT*** (Variable) 3-21***Edit-Fn*** (Variable) B-16***ERROR-OUTPUT*** (Variable) 3-10***Fetch-Definition-Error-Break-Flag*** (Variable) B-16***Getdef-Error-Fn*** (Variable) B-16***Getdef-Fn*** (Variable) B-16***LAST-CONDITION*** (Variable) 3-10***LISPXPRT*** (Property) A-18***NSADDRESS-FORMAT*** (Variable) 4-32***PACKAGE*** (Variable) 3-20,45-46; A-1***PRINT-ARRAY*** (Variable) 3-43***PRINT-BASE*** (Variable) 3-39,42,44***PRINT-BASE*** vs RADIX 3-39***PRINT-CASE*** (Variable) 3-44***PRINT-ESCAPE*** (Variable) 3-41,44***PRINT-LENGTH*** (Variable) 4-22***PRINT-LEVEL*** (Variable) 4-22***PRINT-LEVEL*** & ***PRINT-LENGTH*** vs**PRINTLEVEL** 3-39***PRINT-LEVEL*** or ***PRINT-LENGTH*** is exceeded 3-45***PRINT-RADIX*** (Variable) 3-39,44***READ-BASE*** (Variable) 3-20,44***READ-SUPPRESS*** (Variable) 3-41***READTABLE*** (Variable) 3-39,41-42,48***READTABLE*** vs **SETREADTABLE** 3-39***REMOVE-INTERLISP-COMMENTS*** (Variable) 3-29-30***STANDARD-INPUT*** (Variable) 3-37***STANDARD-INPUT*** vs **INPUT** 3-39***STANDARD-OUTPUT*** (Variable) 3-37***STANDARD-OUTPUT*** vs **OUTPUT** 3-39***Wrap-Parens*** (Variable) B-13***Wrap-Search*** (Variable) B-14**:****:fast-accessors** (Defstruct option) 7-5**:inline** (Defstruct option) 7-5

:template (*Defstruct option*) 7-5

:type (*Defstruct option*) 7-5

=

= (*Event Address*) A-5

?

? (*Exec Command*) A-7

?? (*Exec Command*) A-7

?ACTIVATEFLG (*Variable*) A-24