$LispWorks^{\circ} \mbox{ for the Windows^{\circ} Operating System} \\ LispWorks \ User \ Guide$

Version 4.1



Copyright and Trademarks

LispWorks for the Windows Operating System User's Guide

Version 4.1

November 1998

Part number: 3LBDT2A15NB

Copyright © 1994–1998 by Harlequin Group plc.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Harlequin Group plc.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks is a registered trademark of Harlequin Group plc. Harlequin, Delivery, Transducer/PC, The Authoring Book, ClassWorks, and KnowledgeWorks are all trademarks of Harlequin Group plc.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.

The code for walker.lisp and compute-combination-points is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

CLX and CLUE bear the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Texas Instruments Incorporated, P.O. Box 149149, MS 2151, Austin, Texas 78714-9149

Copyright © 1987, 1988, 1989, 1990, 1991 Texas Instruments Incorporated.

Permission is granted to any individual or institution to use, modify and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and documentation. Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

US Government Use

The LispWorks Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be `unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

http://www.harlequin.com/

Europe:	North America:	Asia Pacific:
Harlequin Limited	Harlequin Incorporated	Harlequin Australia Pty. 1
Parrington Uall	One Cambridge Center	Lovol 19

Level 12 12 Moore Street Canberra, ACT 2601 Australia

telephone +61 2 6206 5522 fax +61 2 6206 5525

Ltd.

Barrington Hall Barrington Cambridge CB2 5RG UK

telephone +44 1223 873 800 fax +44 1223 873 873 One Cambridge Center Cambridge, MA 02142 USA

telephone +1 617 374 2400 fax +1 617 252 6505

Contents

1 Introduction 1

About this manual 1 The LispWorks manuals 2 Other documentation 4 Notation 4

2 The Debugger 5

Entering the TTY debugger 6 Simple use of the TTY debugger 7 The stack in the debugger 8 TTY debugger commands 8 Debugger control variables 16

3 The TTY Inspector 19

Describe 19 Inspect 20 Inspection modes 22

4 The Trace Facility 27

Simple tracing 27 Tracing options 29 Example 36 Tracing methods 37 Trace variables 37

5 The Advice Facility 41

Combining the advice 43 Removing advice 44 Advice for macros and methods 45 Example 47 Advice functions and macros 50

6 Action Lists 51

Defining and undefining action lists 52 Exception handling variables 54 Other variables 55 Diagnostic utilities 55 Examples 56

7 The Compiler 59

Compiling a function 60 Compiling a source file 61 How the compiler works 61 Compiler control 62 Declare, proclaim, and declaim 66 Compiler parameters affecting LispWorks 67

8 Storage Management 69

Introduction 69 Generations and segments 70 Allocation of objects to generations 71 Allocation of static objects 71 Allocation in different generations 72 Mark and sweep 72 Promotion 73 Garbage collection strategy 73 Overflow 74 Behavior of generation 1 74 Behavior of generation 2 75 Timing the garbage collector 75 Summary of garbage collection symbols 75

9 The Profiler 77

What the profiler does 77 Setting up the profiler 78 Running the profiler 79 Interpretation of profiling results 79 Profiling pitfalls 80 Profiling and garbage collection 81

10 Simple Customization of LispWorks 83

Customizing the editor 84 Using ! for :redo 85 Customizing LispWorks for use with your own code 85

11 Multiprocessing 87

Introduction to processes 87 The process programming interface 88 Locks 90 Example 91

12 Common Defsystem 93

Introduction 93 Defining a system 94

13 The Parser Generator 101

Introduction 101 Grammar rules 101 Functions defined by defparser 103 Error handling 104 Interface to lexical analyzer 105 Example 105

14 Dynamic Data Exchange 109

Introduction 109 Client interface 111 Server interface 117

15 Common SQL 123

Introduction 123 Initialization 125 Functional interface 129 Object oriented interface 138 Symbolic SQL syntax 143 SQL I/O recording 151 SQL errors 152

16 User Defined Streams 155

Introduction 155 An illustrative example of user defined streams 155

Index 165

1

Introduction

1.1 About this manual

This manual describes the features and tools available in LispWorks, and how to use them.

The first group of chapters in this book describes some of the central programming tools and features in LispWorks:

- Chapter 2, "The Debugger", covers the TTY debugger.
- Chapter 3, "The TTY Inspector".
- Chapter 4, "The Trace Facility".
- Chapter 5, "The Advice Facility".
- Chapter 6, "Action Lists".
- Chapter 7, "The Compiler".
- Chapter 8, "Storage Management", covers the use of the garbage collector.
- Chapter 9, "The Profiler".

The next chapter, Chapter 10, "Simple Customization of LispWorks", explains how to perform some commonly required customizations, such as controlling start-up appearance of LispWorks. The next group of chapters describe features of specialist interest.

- Chapter 11, "Multiprocessing", including locks.
- Chapter 12, "Common Defsystem" describes how to use defsystem to combine a series of source files into a manageable project.
- Chapter 13, "The Parser Generator".
- Chapter 14, "Dynamic Data Exchange" describes how to provide Dynamic Data Exchange functionality to your applications.
- Chapter 15, "Common SQL" explains how to use LispWorks to communicate with ODBC databases using SQL.
- Chapter 16, "User Defined Streams" provides an illustrative example showing how to define and implement your own streams.

Please note that documentation for Graphics Ports has been moved to the LispWorks *CAPI User Guide* and LispWorks *CAPI Reference Manual.l*

1.2 The LispWorks manuals

The LispWorks manual set comprises the following books:

- The *LispWorks User Guide*—this book—describes the features and tools available in LispWorks.
- The *LispWorks Reference Manual* contains detailed information on all functions, macros, variables and classes available in LispWorks, in alphabetical order.
- The *Common LispWorks User Guide* describes Common LispWorks, the user interface for LispWorks. Common LispWorks is a set of windowing tools that let you develop and test Common Lisp code more easily and quickly.
- The LispWorks *Editor User Guide* describes the keyboard commands and programming interface to the Common LispWorks editor tool.
- The LispWorks *CAPI User Guide* and the LispWorks *CAPI Reference Manual* describe the CAPI. This is a library of classes, functions, and macros for developing graphical user interfaces for your applications.

The LispWorks *CAPI User Guide* is a tutorial guide to the CAPI, and the LispWorks *CAPI Reference Manual* is an in-depth reference text.

- The LispWorks *Foreign Language Interface User Guide and Reference Manual* explains how you can use C source code in applications developed using LispWorks.
- The LispWorks *Delivery User Guide* describes how you can deliver working, standalone versions of your LispWorks applications for distribution to your customers.
- The LispWorks *KnowledgeWorks and Prolog User Guide* describes the LispWorks toolkit for building knowledge-based systems. Prolog is a logic programming system within Common Lisp.
- The *CommonLisp Interface Manager 2.0 User's Guide* describes the portable Lisp-based GUI toolkit.
- A set of installation notes explain how to install LispWorks and start it running, also containing a set of release notes is provided that documents last minute issues that could not be included in the main manual set.

These books are all available in online form, in both HTML format and PDF format.

Commands in the **Help** menu of any of the Common LispWorks tools give you direct access to the online documentation in HTML format, using the HTML browser that is supplied with LispWorks. Details of how to use these commands can be found in the *Common LispWorks User Guide*.

Documentation is also provided in PDF form. You can use Adobe Acrobat Reader to browse the PDF documentation online or to print it. Adobe Acrobat Reader is available for free download from Adobe's web site, http://www.adobe.com/.

Please let us know if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.

1.3 Other documentation

The LispWorks manuals do not attempt to describe Lisp itself. For definitive information on Common Lisp, including CLOS, consult the American National Standard X3.226 for Common Lisp. An HTML version of this document is supplied with LispWorks and can be accessed from the Help menu of any Common LispWorks tool. For information on CLOS, Sonya E. Keene's book *Object-Oriented Programming in Common Lisp: A Programmers' Guide* is very helpful. This book is published by Addison-Wesley. For an account of Meta-Object protocols as well as a detailed study of an implementation of CLOS see Kiczales, Rivieres and Bobrow, *The Art of the Meta-Object Protocol*, published by MIT Press.

1.4 Notation

The LispWorks manuals follow the notation used in *Common Lisp: the Language* (2nd Edition).

Please note that your windows may differ in some respects from the illustrations given in the LispWorks manuals. This is because some details are controlled by the window manager that you are using, not by LispWorks itself.

The Debugger

The debugger is an interactive tool for examining and manipulating the Lisp environment. Within the debugger you have access to not only the interpreter, but also to a variety of debugging tools. The default behavior when any error occurs is to enter the debugger. Users can then trace backwards through the history of function calls to determine how the error arose. They may inspect and alter local variables of the functions on the execution stack, and possibly continue execution by invoking a pre-defined restart (if available) or by forcing any function invocation on the stack to return user-specified values.

When writing an application it is possible to prevent entry to the debugger when an error occurs, by creating condition handlers to take some appropriate action to recover without user intervention. It is also possible to use restarts to specify some default methods of error recovery. The debugger is entered whenever an error is signalled (via a call to error or cerror) and not handled by an error handler, or it can be explicitly invoked via a call to break.

You can use the debugger in TTY mode (that is, from the listener command line) or using the dedicated debugger tool in the Common LispWorks environment. This chapter describes the TTY debugger; please refer to the *Common LispWorks User Guide* for details about the debugger tool.

The compiler generates information necessary for the use of the debugger during compilation. Users can opt for faster compilation, at the expense of reducing the information available to the debugger. See under togglesource-debugging in the *LispWorks Reference Manual*.

2.1 Entering the TTY debugger

The following is a simple example.

The call to error causes entry into the debugger. The final prompt in the example contains a 1 to indicate that the top level of the debugger has been entered. The debugger can be entered recursively, and the prompt shows the current level. Once inside the debugger, you may use all the facilities available at the top-level in addition to the debugger commands.

The debugger may also be invoked by using the trace facility to force a break at entry to or exit from a particular function, or by a keyboard interrupt (generated by Ctrl+Break).

2.2 Simple use of the TTY debugger

Upon entering the debugger as a result of an error, a message describing the error is printed and a number of options to continue (called restarts) are presented. Thus:

```
CL-USER 6 > (/ 3 0)
Error: Division-by-zero caused by / of (3 0)
1 (continue) Return a value to use
2 Supply new arguments to use
3 (abort) return to level 0.
4 return to top loop level 0.
5 Destroy process.
Type :c followed by a number to proceed
CL-USER 7 : 1 >
```

To select one of these restarts, enter :c (continue) followed by the number of the restart. So in the above example you could continue as follows:

```
CL-USER 7 : 1 > :c 2
Supply first number: 33
Supply second number: 11
3
CL-USER 8 >
```

There are two special restarts, a continue restart and an abort restart. These are indicated by the bracketed word continue or abort at their start. The continue restart can be invoked by typing :c alone. Similarly, the abort restart can be invoked by entering :a. So an alternative continuation of the division example would be:

```
CL-USER 7 : 1 > :c
Supply a form to be evaluated and used: (+ 4 5)
9
```

2.3 The stack in the debugger

The debugger allows you to examine the state of the execution stack. This consists of a sequence of frames representing active function invocations, special variable bindings, restarts, active catchers, active handlers and system-related code. In particular the execution stack has a call frame for each active function call (that is for each function that has been entered but from which control has not yet returned). The top of the stack contains the most recently created frames (and so the innermost calls), and the bottom of the stack contains the oldest frames (and so the outermost calls). You can examine a call frame to find the function's name, and the names and values of its arguments.

Catch frames are established by using the special form catch, and exist to receive throws to the matching tag. Restart frames correspond to restarts that have been set up, and handler frames correspond to the error handlers currently active. Binding frames are formed when special variables are bound. Open frames are established by the system. By default only the catch frames and the call frames are displayed. However the remaining types of frame are displayed if you set the appropriate variables (see Section 2.5 on page 16).

Within the debugger there are commands to examine a stack frame, and to move around the stack. These are explained in the following section. Typing :help in the debugger also produces a command listing.

2.4 TTY debugger commands

This section describes commands specific to the debugger. In addition to these commands the top-level loop interpreter can be used as normal. Upon entry to the debugger the implicit current stack frame is set to the top of the execution stack. The commands allow you to move around the stack, to examine the current frame, and to leave the debugger. The commands are all keywords, and as such case-insensitive, but are shown here in lower case for clarity.

2.4.1 Backtracing

A backtrace is a list of the stack frames starting at the current frame and continuing down the stack. The backtrace thus displays the sequence by which the functions were invoked, starting with the most recent. For instance:

```
CL-USER 10 > (defun function-1 (a b c))
               (function-2 (+ a b) c))
FUNCTION-1
CL-USER 11 > (defun function-2 (a b)
               (function-3 (+ a b)))
FUNCTION-2
CL-USER 12 > (defun function-3 (a) (/ 3 (- 111 a)))
FUNCTION-3
CL-USER 13 > (function-1 1 10 100)
Error: Division-by-zero caused by / of (3 0)
  1 (continue) Return a value to use
  2 Supply new arguments to use
  3 (abort) return to level 0.
  4 return to top loop level 0.
  5 Destroy process.
Type :c followed by a number to proceed
CL-USER 14 : 1 > :bq 10
SYSTEM::DIVISION-BY-ZERO-ERROR <- / <- FUNCTION-3
<- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- FUNCTION-2
<- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- FUNCTION-1
<- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- SYSTEM::%INVOKE <-
SYSTEM::%EVAL
CL-USER 15 : 1 >
```

In the above example the command to show a quick backtrace was used (:bq). Instead of showing each stack frame fully, this only shows the function name associated with each of the call frames. The number 10 following :bq specifies that only the next ten frames should be displayed rather than continuing to the bottom of the stack.

Debugger command

:b &optional verbose m

:b

This is the command to obtain a backtrace from the current frame. It may optionally be followed by :verbose, in which case a fuller description of each frame is given that includes the values of the arguments to

the function calls. It may also be followed by a number (*m*), specifying that only that number of frames should be displayed.

Debugger command

:bq m

:bq

This produces a quick backtrace from the current position. Only the call frames are included, and only the names of the associated functions are shown. If the command is followed by a number then only that many frames are displayed.

2.4.2 Moving around the stack

On entry to the debugger the current frame is the one at the top of the execution stack. There are commands to move to the top and bottom of the stack, to move up or down the stack by a certain number of frames, and to move to the frame representing an invocation of a particular function.

:>	Debugger command	
	This sets the current frame to the one at the bottom of the stack.	
:<	Debugger command	
	This sets the current frame to the one at the top of the stack.	
:p	Debugger command	
	:p [<i>m</i> <i>fn</i> - <i>name</i>]	
	By default this takes you to the previous frame on the stack. If it is fol- lowed by a number then it moves that number of frames up the stack. If it is followed by a function name then it moves to the previous call frame for that function.	
:n	Debugger command	

:n [m|fn-name]

Similar to the above, this goes to the next frame down the stack, or *m* frames down the stack, or to the next call frame for the function *fn-name*.

2.4.3 Miscellaneous commands

:V

Debugger command

This displays information about the current stack frame. In the case of a call frame corresponding to a compiled function the names and values of the function's arguments are shown. For an interpreted function the names and values of local variables are also given. If the value of an argument is not known (perhaps because the code has been compiled for speed rather than other considerations), then it is printed as the keyword :dont-know.

Debugger command

This reprints the message which was displayed upon entry to the current level of the debugger. This is typically an error message and includes several continuation options.

Debugger command

:cc &optional *var*

This returns the current condition object which caused entry to this level of the debugger. If an optional *var* is supplied then this must be a symbol, whose symbol-value is set to the value of the condition object.

Debugger command

This allows you to edit the function associated with the current frame. If you are using TAGS, you are prompted for a TAGS file.

Debugger command

:all &optional flag

:error

:CC

:ed

:all

This option enables you to set the debugger option to show all frames (if *flag* is non-nil), or back to the default (if *flag* is nil). By default, *flag* is t.

:lambda

:redo

Debugger command

This returns the lambda expression for an anonymous interpreted frame. If the expression is not known, then it is printed as the keyword :dont-know

Debugger command

:redo &optional command-identifier

This option repeats a previous command. The *command-identifier* is either a number in the history list or a substring of the command. This command also works in the listener, allowing you to work with Common Lisp forms easily.

Debugger command

:get name command-identifier

A call to :get retrieves a command from the history list and places it in the variable *name*. The *command-identifier* is the history list number of the command to be retrieved. This command also works in the listener, allowing you to work with Common Lisp forms easily.

Debugger command

:use new-form form-to-replace &optional command-identifier

This option replaces *form-to-replace* with *new-form* in a previous command and repeats that command. If supplied, *command-identifier* is the history list number of the command you want to repeat. If *commandidentifier* is not supplied, the last command is repeated by default. This command also works in the listener, allowing you to work with Common Lisp forms easily.

:get

:use

Debugger command

This option produces a list of the command history. This command also works in the listener, allowing you to work with Common Lisp forms easily.

Debugger command

This command prints symbols from other packages corresponding to the symbol that was called, but could not be found, in the current package. Any such symbols are also offered as restarts when you first enter the debugger.

NEW 21 > (initialize-graphics-port)

Error: Undefined function INITIALIZE-GRAPHICS-PORT called with arguments (). 1 (continue) Try invoking INITIALIZE-GRAPHICS-PORT again. 2 Return some values from the call to INITIALIZE-GRAPHICS-PORT.

3 Try invoking GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT with the same arguments.

4 Set the symbol-function of INITIALIZE-GRAPHICS-PORT to the symbol-function of GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT.

5 Try invoking something other than INITIALIZE-GRAPHICS-PORT with the same arguments.

6 Set the symbol-function of INITIALIZE-GRAPHICS-PORT to another function.

7 (abort) Return to level 0.

8 Return to top loop level 0.

Type :c followed by a number to proceed or type :? for other options

```
NEW 22 : 1 > :lf
Possible candidates are (GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT)
GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT
```

NEW 23 : 1 >

:his

:If

2.4.4 Leaving the debugger

You may leave the debugger either by taking one of the continuation options initially presented, or by explicitly specifying values to return from one of the frames on the stack.

Debugger command

This selects the **:abort** option from the various continuation options that are displayed when you enter the current level of the debugger.

Debugger command

:c &optional m

If this is followed by a number then it selects the option with that number, otherwise it selects the :continue option.

Debugger command

:ret

:a

:C

:ret value

This causes *value* to be returned from the current frame. It is only possible to use this command when the current frame is a call frame. Multiple values may be returned by using the *values* function. So to return the values 1 and 2 from the current call frame, you could type

```
:ret (values 1 2)
```

:res

Debugger command

:res m

Restarts the current frame. If *m* is non-nil, you are prompted for new arguments. If *m* is nil, the original arguments to the frame are used.

:top

Debugger command

Aborts to the top level of the debugger.

2.4.5 Example TTY session

This section presents a short interactive debugging session. It starts by defining a routine to calculate Fibonacci Numbers, and then erroneously calls it with a string.

1. First, define the fibonacci function shown below in a listener.

2. Next, call the function as follows.

(fibonacci "turtle")

The system generates an error, since = expects its arguments to be numbers, and displays several continuation options, so that you can try to find out how the problem arose.

3. Type :bq at the debugger prompt to perform a quick backtrace. Notice that the problem is in the call to fibonacci.

Note that the calls to *%apply-interpreted-function in the backtrace occur because fibonacci is being interpreted.

You should have passed the length of the string as an argument to fibonacci, rather than the string itself.

4. Do this now, by typing the following form at the debugger prompt.

```
(legnth "turtle")
```

You intended to call fibonacci with the length of the string, but typed in length incorrectly. This takes you into the second level of the debugger. Note that the continuation options from your entry into the top level of the debugger are still displayed, and are listed after the new options. You can select any of these options.

5. Type :a to return to the top level of the debugger.

6. Type : v to display variable information about the current stack frame in the debugger.

The following output is displayed:

```
M : "turtle"
INDEX : 2
FIB-N-2 : 1
FIB-N-1 : 1
```

You need to set the value of the variable **m** to be the length of the string "turtle", rather than the string itself.

7. Type in the form below.

(setq m (length "turtle"))

In order to get the original computation to resume using the new value of m, you still need to handle the original error.

8. Type **:error** to remind yourself of the original error condition you need to handle.

You can handle this error by returning mil from the call to =, which is the result that would have been obtained if m had been correctly set.

- **9.** Type **:**c to invoke the continue restart, which in this case requires you to return a value to use.
- 10. When prompted for a form to be evaluated, type nil.

This causes execution to continue as desired, and you can obtain the final result with no further problems.

2.5 Debugger control variables

common-lisp:*debug-io*

Variable

The value of this variable is the stream which the debugger uses for its input and output.

dbg:*debug-print-length*

The value to which common-lisp:*print-length* is bound during output from the debugger.

dbg:*debug-print-level*

The value to which common-lisp:*print-level* is bound during output from the debugger.

dbg:*hidden-packages*

This variable should be bound to a list of packages. The debugger suppresses symbols from these packages (so, for example, it does not display call frames for functions in these packages).

dha	ı.*n	rint	hir	ding	n fra	moc*
ubu	1: P	1111	ווע-	iunių	J-11 a	mes*

dbg:*print-catch-frames*

dbg:*print-handler-frames*

dbg:*print-open-frames*

dbg:*print-restart-frames*

dbg:*print-non-symbol-frames*

These six variables control whether or not the corresponding types of frame are displayed by the debugger. For each variable, if the variable is non-nil then that type of frame is shown. Initially only dbg:*printcatch-frames* is non-nil. Note that the call frames are always displayed.

The following function is used in conjunction with these variables.

dbg:set-debugger-options

dbg:set-debugger-options &key all bindings catchers hidden non-symbol handler restarts invisible

Variable

Variables

Function

Variable

Variable

A call to set-debugger-options allows you to set the above variables without having the inconvenience of setting each variable individually with a call to setq and without having to remember the precise names for each of the variables.

The keywords in the function refer to the different system variables as described below:

:all — affects the state of the :all command. :bindings — dbg:*print-binding-frames* :catchers — dbg:*print-catch-frames* :hidden — dbg:*hidden-packages* :non-symbol — dbg:*print-non-symbol-frames* :handler — dbg:*print-handler-frames* :restarts — dbg:*print-restart-frames* :invisible — dbg:*print-invisible-frames*

The TTY Inspector

LispWorks provides two inspectors. One is for use with the Common LispWorks environment, and is described in the *Common LispWorks User Guide*. The other is the TTY inspector, which uses a stream interface, and can be used on any terminal (in particular within a Common LispWorks listener). Both inspectors allow you to traverse complex data structures interactively and to destructively modify components of these structures. However, the two inspectors are quite different. No attempt has been made to make their usage compatible and instead each inspector is designed to exploit to the full the particular environment facilities available.

The teletype inspector provides a simple inspector facility which can be used on a stream providing line breaks as the only type of formatting. It is built on top of the describe function which is briefly described below and modifies the top level loop in a similar way to the debugger (see Chapter 2, "The Debugger").

3.1 Describe

The function describe displays the slots of composite data structures in a manner dependent on the type of the object. The slots are labeled with a name where appropriate, or otherwise with a number.

The example below shows the result of calling describe on a simple list.

describe describes slots recursively up to a limit set by the special variable system::*describe-level*. Note that only arrays, structures and conses are printed recursively. The slots of all other object types are only printed when at the top level of describe.

```
system::*describe-level* has an initial value of 2.
```

The symbols system::*describe-print-level* and system::*describeprint-length* are similar in effect to hcl:*trace-print-level* and hcl:*trace-print-length*. They control, respectively, the depth to which nested objects are printed (initial value 10), and the number of components of an object which are printed (initial value 10).

To customize describe, define new methods on the generic function describe-object.

3.2 Inspect

The function inspect is an interactive version of describe. It displays objects in a similar way to describe. Entering the teletype inspector causes a new level of the top loop to be entered with a special prompt indicating that the inspector has been entered and showing the current inspector level.

In the modified top loop, if you enter a slot name, that slot is inspected and the current object is pushed onto an internal stack of previously inspected objects. The special variables \$, \$\$, and \$\$\$ are bound to the top three objects on the inspector stack.

The following keywords are also treated specially by the inspector.

Table 3.1 Inspector keywords

:d	Display current object.	
:dm	Display more of current object.	
:i m	Recursively invoke a new inspector. <i>m</i> is an object to inspect.	
٠q	Quit current inspector.	
:s <i>n</i> v	Sets slot <i>n</i> to value <i>v</i> .	
:sh	Show inspector stack.	
:u int	Undo last inspection. If you supply an optional integer argument, <i>int</i> , then the last <i>int</i> inspections are undone.	
:ud	Undo last inspection and redisplay current object.	
:m	Change the inspection mode — see Section 3.3 on page 22.	

The variables sys::*inspect-print-level* and sys::*inspect-printlength* are similar to sys::*describe-print-level* and sys::*describeprint-length* (see above).

:dm displays more slots of the current object. If the object has more than sys::*describe-length* slots, then the first *describe-length* will be printed, followed by an ellipsis and then

(:dm for more)

If you type :dm at the prompt, the next *describe-length* slots are displayed. This only works on the last inspected object, so if you recursively inspect a slot and come back, :dm does not do anything useful. Typing :d lets you view the object again.

:ud is equivalent to typing :u followed by :d.

3.3 Inspection modes

The :m command displays and changes the current inspection mode for an inspected value. The session below demonstrates how it works:

```
CL-USER 1 > (inspect "a
string with
newlines in it")
"a
string with
newlines in it" is a STRING
[0] : "a"
[1] : "string with"
[2] : "newlines in it"
CL-USER 2 : Inspect 1 > :m
   1. STRING
* 2. LINES
CL-USER 3 : Inspect 1 >
```

The :m produces an enumerated list of inspection modes for this value.

The asterisk next to

* 2. LINES

means that LINES is the current inspection mode.

You can change mode by typing :m followed by the name or number of the mode to which you are changing:

```
CL-USER 3 : Inspect 1 > :m 1
"a
string with
newlines in it" is a STRING
[0]
      : #\a
      : #\Newline
[1]
[2]
     : #\s
      : #\t
[3]
[4]
      : #\r
      : #\i
[5]
[6]
     : #\n
[7]
      : #\g
[8]
     : #\Space
     : #\w
[9]
[10] : #\i
    : #\t
[11]
    : #\h
[12]
    : #\Newline
[13]
[14]
    : #\n
[15] : #\e
[16]
    : #\w
[17] : #\1
    : #\i
[18]
[19] : #\n ..... (:dm for more)
```

CL-USER 4 : Inspect 1 >

3.3.1 Hash table inspection modes

There are three hash table inspection modes. They can be accessed in either the Common LispWorks or teletype inspectors.

A brief introduction to the representation of hash tables is necessary so that you can fully understand what you gain from the new modes.

Internally, a hash table is a structure containing, among other things,

- a big vector
- size and growth information
- accessing functions.

When keys and values are added to the table, sufficiently similar keys are converted into the same index in the vector. When this happens, the similar keys and values are kept together in a chain that hangs off this place in the vector.

The different inspection modes provide views of different pieces of this structure:

HASH-TABLE This mode is the "normal" view of a hash table; as a table of keys and values. When you inspect an item you inspect the value of the item.

ENUMERATED-HASH-TABLE

This mode is a variation of the normal view, where a hash table is viewed simply as a list of lists. When you inspect an item you are inspecting a list containing a key and a value.

HASH-TABLE-STATISTICS

This mode shows how long the chains in the hash table are, so that you can tell how efficiently it is being used. For example, if all chains contained fewer than two items the hash table would be being used well.

HASH-TABLE-HISTOGRAM

This mode shows the statistical information from HASH-TABLE-STATISTICS as a histogram.

STRUCTURE This mode provides a raw view of the whole hash table structure. When you inspect an item you are inspecting the value of that slot in the hash table structure.

Here is an example of hash table inspection.

```
CL-USER 1 > (setq hash (make-hash-table))
Warning: Setting unbound variable HASH
#<EQL Hash table{0} 49ee564>
CL-USER 2 > (setf (gethash 'lisp hash) 'programming
                  (gethash 'prolog hash) 'programming
                  (gethash 'c hash) 'programming
                  (gethash 'c++ hash) 'programming
                  (gethash 'english hash) 'natural
                  (gethash 'german hash) 'natural)
NATURAL
CL-USER 3 > (inspect hash)
#<EQL Hash table{6} 49ee564> is a HASH-TABLE
          : PROGRAMMING
[C++]
          : NATURAL
[GERMAN]
[PROLOG]
          : PROGRAMMING
[LISP]
          : PROGRAMMING
[ENGLISH] : NATURAL
[C]
           : PROGRAMMING
CL-USER 4 : Inspect 1 > :m
  1. HASH-TABLE
  2. HASH-TABLE-HISTOGRAM
  3. HASH-TABLE-STATISTICS
  4. ENUMERATED-HASH-TABLE
  5. STRUCTURE
CL-USER 5 : Inspect 1 > :m 3
[chain of length 0 :]
                        : 25
[chain of length 1 :]
                      : 6
CL-USER 6 : Inspect 1 >
```

The hash table statistics show that hash has 31chains, of which 25 are empty and 6 have one entry.

Here, the same information is represented as a histogram.

Below is the raw representation of the hash table:

```
CL-USER 7 : Inspect 1 > :m 5
#<EQL Hash table{6} 49ee564> is a HASH-TABLE
[KIND]
                               : EQL
[SIZE]
                              : 31

      [SIZE]
      : 31

      [REHASH-SIZE]
      : 2.0

      [REHASH-THRESHOLD]
      : 1.0

      [THRESHOLD]
      : 31

      [COUNTER]
      : 850

                              : 850
[COUNTER]
[NUMBER-ENTRIES] : 6
[TABLE]
                              : #(NIL NIL NIL NIL
                               #%((C . PROGRAMMING) NIL) NIL
                               NIL #%((ENGLISH . NATURAL) NIL)
                               NIL NIL ...)
[NO-DESTRUCT-REHASH] : NIL
                             : 5
[POWER2]
                             : SYSTEM::DIVIDE-5-1
[HASH-REM]
[HASH-FN]
                              : SYSTEM::EQL-HASHFN
                      : SYSTEM::EQL-HASHFN
: SYSTEM::GETHASH-EQL
: SYSTEM::PUTHASH-EQL
: SYSTEM::REMHASH-EQL
[GETHASH-FN]
[PUTHASH-FN]
[REMHASH-FN]
[ENSURE-TLATTER-FN] : SYSTEM::ENSURE-TLATTER-EQL
```

CL-USER 8 : Inspect 1 >

The Trace Facility

The trace facility is a debugging aid enabling you to follow the execution of particular functions. At any time there are a set of functions (and macros and methods) which are being monitored in this way. The normal behavior when a call is made to one of these functions is for the function's name, arguments and results to be printed out by the system. More generally you can specify that particular forms should be executed before or after entering a function, or that certain calls to the function should cause it to enter the main debugger. Tracing of a function continues even if the function is redefined; however the tracing of some structure accessors and so forth may be lost if the compiler is set to optimize the code for efficiency (so that these calls are inlined).

The standard way of getting functions to be traced in this way is to call the macro trace with the symbols of the functions (or macros or generic functions) concerned. In addition it is possible to restrict tracing to a particular method (rather than a generic function), by specifying the requisite classes for the arguments in the call to trace. The trace facility handles recursive and nested calls to the functions concerned.

4.1 Simple tracing

This section shows you how to perform simple traces.

1. Type this definition of the factorial function fac into the listener:

```
(defun fac (n)
(if (= n 1) 1
    (* n (fac (- n 1)))))
```

2. Now trace the function by typing the following into the listener.

(trace fac)

3. Call the function fac as follows:

(fac 3)

The following trace output appears in the listener.

```
0 FAC > (3)

1 FAC > (2)

2 FAC > (1)

2 FAC < (1)

1 FAC < (2)

0 FAC < (6)
```

Upon entry to each traced function call, trace prints the following information:

- The *level* of tracing, that is, the number of recursive entries to trace (starting at 0).
- The function name.
- The argument for the current call.

Each call is indented according to the level of tracing for the call.

Upon exit from each call, the same information is produced: The > symbol denotes entry to a function, and the < symbol denotes exit from it.

Output produced in this way is always sent to a special stream, *traceoutput*, which is either associated with the listener, or with background output. You can give other expressions to be sent to this stream, in addition to the arguments and results of a function.

Calling trace with no arguments produces a list of all the functions currently being traced. In order to cease tracing a function the macro untrace should be called with commands. All tracing can be removed by calling untrace with no arguments.

```
CL-USER 5 > (untrace fac)
NIL
CL-USER 6 > (fac 4)
24
CL-USER 7 >
```

4.2 Tracing options

There are a number of options available when using the trace facilities, which allow you both to restrict or expand upon the information printed during a trace. For instance, you can restrict tracing of a function to a particular process, or specify additional actions to be taken on function call entry and exit.

Note that the options and values available only apply to a particular traced function. Each traced function has its own, independent, set of options.

This section describes the options that are available. Each option can be set as described above.

4.2.1 Evaluating forms on entry to and exit from a traced function

:before

Trace keyword

:before list of forms

If non-nil, the list of forms is evaluated on entry to the function being traced. The forms are evaluated and the results printed after the arguments to the function.

Here is an example of its use. *traced-arglist* is bound to the list of arguments given to the function being traced. In this example, it is used to accumulate a list of all the arguments to fac across all iterations.

1. In the listener, initialize the variable args-in-reverse as follows:

(setq args-in-reverse ())

 For the fac function used earlier, set the value of :before to the following list:

((push (car *traced-arglist*) args-in-reverse))

3. In the listener, evaluate the following form:

(fac 3)

After evaluating this form, args-in-reverse has the value (1 2 3), that is, it lists the arguments which fac was called with, in the reverse order they were called in.

:after

Trace keyword

:after list of forms

If non-nil, this option evaluates a list of forms upon return from the function to be traced. The forms are evaluated and the results printed after the results of a call to the function.

This option is used in exactly the same way as :before. For instance, using the example for :before as a basis, create a list called results-inreverse, and set the value of :after so that (car *traced-results*) is pushed onto this list. After calling fac, results-in-reverse contains the results returned from fac, in the reverse order they were called in.

Note: *traced-arglist* is still bound as well.

4.2.2 Evaluating forms without printing results

:eval-before

Trace keyword

:eval-before list-of-forms

This option allows you to supply a list of forms for evaluation upon entering the traced function. The forms are evaluated after printing out the arguments to the function, but unlike :before their results are not printed.

:eval-after

Trace keyword

:eval-after list-of-forms

This option allows you to supply a list of forms for evaluation upon leaving the traced function. The forms are evaluated after printing out the results of the function call, but unlike **:after** their results are not printed.

4.2.3 Using the debugger when tracing

:break

Trace keyword

:break form

If form evaluates to non-nil, the debugger is entered directly from trace. If it returns nil, tracing continues as normal. This option lets you force entry to the debugger by supplying a form as simple as t.

Upon entry to the traced function, the standard trace information is printed, any supplied :before forms are executed, and then *form* is evaluated.

:break-on-exit

Trace keyword

:break-on-exit form

Like :break, this option allows you to enter the debugger from trace. It differs in that the debugger is entered *after* the function call is complete.

Upon exit from the traced function, the standard trace information is printed, and then *form* is evaluated. Finally, any supplied **:after** forms are executed.

:backtrace

Trace keyword

:backtrace backtrace

Generates a backtrace on each call to the traced function. *backtrace* can be any of the following values:

:quick	Like the :pd debugger command.
t	Like the :ь debugger command.
:verbose	Like the :b :verbose debugger command.
:bug-form	Like the :bug-form debugger command.

4.2.4 Entering stepping mode

:step

Trace keyword

:step form

When non-nil, this option puts the trace facility into stepper mode, where interpreted code is printed out one step of execution at a time.

4.2.5 Configuring function entry and exit information

:entrycond

Trace keyword

:entrycond form

This option controls the printing of information on entry to a traced function. *form* is evaluated upon entry to the function, and information is printed if and only if *form* evaluates to t. This allows you to turn off printing of function entry information by supplying a *form* of nil, as in the example below.

:exitcond

Trace keyword

:exitcond form

This option controls the printing of information on exit from a traced function. *form* is evaluated upon exit from the function, and, like :entrycond, information is printed if and only if *form* evaluates to t. This allows you to turn off printing of function exit information by supplying a *form* of nil.

An example of using :exitcond and :entrycond is shown below:

1. For the fac function, set the values of :entrycond and :exitcond as follows.

```
:entrycond => (evenp (car *traced-arglist*))
:exitcond => (oddp (car *traced-arglist*))
```

Information is only printed on entry to fac if the argument passed to fac is even. Conversely, information is only printed on exit from fac if the argument passed to fac is odd.

2. Type the following call to fac in a listener:

CL-USER 12 > (fac 10)

The tracing information printed is as follows:

```
0 FAC > (10)

2 FAC > (8)

4 FAC > (6)

6 FAC > (4)

8 FAC > (2)

9 FAC < (1)

7 FAC < (6)

5 FAC < (120)

3 FAC < (5040)

1 FAC < (362880)
```

4.2.6 Directing trace output

:trace-output

Trace keyword

:trace-output stream

This option allows you to direct trace output to a stream other than the listener in which the original function call was made. By using this you can arrange to dispatch traced output from different functions to different places.

Consider the following example:

1. In the listener, create a file stream as follows:

```
CL-USER 129 > (setq str (open "trace.txt" :direction :output))
Warning: Setting unbound variable STR
#<File stream "/u/neald/trace.txt">
```

- 2. Set the value of the :trace-output option for the function fac to str.
- 3. Call the fac function, and then close the file stream as follows:

```
CL-USER 138 > (fac 8)
40320
CL-USER 139 > (close str)
T
```

Inspect the file trace.txt in order to see the trace output for the call of (fac 8).

4.2.7 Restricting tracing

:process

Trace keyword

:process process

This lets you restrict tracing of a function to a particular process. If *process* evaluates to t, then the function is traced from within all processes (this is the default). Otherwise, the function is only traced from within the process that *process* evaluates to.

:when

Trace keyword

:when form

This lets you invoke the tracing facilities on a traced function selectively. Before each call to the function, *form* is evaluated. If *form* evaluates to nil, no tracing is done. The contents of hcl:*traced-arglist* can be examined by *form* to find the arguments given to trace.

4.2.8 Storing the memory allocation made during a function call

:allocation

Trace keyword

:allocation form

If *form* is non-nil, this prints the memory allocation, in bytes, made during a function call. The symbol that *form* evaluates to is used to accumulate the amount of memory allocated between entering and exiting the traced function.

Note that this symbol continues to be used as an accumulator on subsequent calls to the traced function; the value is compounded, rather than over-written.

Consider the example below:

- 1. For the fac function, set the value of :allocation to \$\$fac-alloc.
- 2. In the listener, call fac, and then examine the value of \$\$fac-alloc.

```
CL-USER 152 > $$fac-alloc
744
```

4.2.9 Tracing functions from inside other functions

:inside

Trace keyword

:inside list-of-functions

The functions given in the argument to **:inside** should reference the traced function in their implementation. The traced function is then only traced in calls to any function in the list of functions, rather than in direct calls to itself.

For example:

1. Define the function fac2, which calls fac, as follows:

```
(defun fac2 (x)
  (fac x))
```

- 2. For the fac function, set the value of :inside to fac2.
- 3. Call fac, and notice that no tracing information is produced.

```
CL-USER 154 > (fac 3)
6
```

4. Call fac2, and notice the tracing information.

```
CL-USER 155 > (fac2 3)

0 FAC > (3)

1 FAC > (2)

2 FAC > (1)

2 FAC < (1)

1 FAC < (2)

0 FAC < (6)
```

4.3 Example

The following example illustrates how trace may be used as a debugging tool. Suppose that you have defined a function \pm , and intend its first argument to be a non-negative number. You can trap calls to \pm where this is not true, providing an entry into the main debugger in these cases. It is then possible for you to investigate how the problem arose.

To do this, you specify a :break option for f using trace. If the form following this option evaluates to a non-nil value upon calling the function, then the debugger is entered. In order to inspect the first argument to the function f, you have access to the variable *traced-arglist*. This variable is bound to a list of the arguments with which the function was called, so the first member of the list corresponds to the first argument of f when tracing f.

```
CL-USER 12 > (defun f (a1 a2) (+ (sqrt a1) a2))
F
CL-USER 13 > (trace (f :break (< (car *traced-arglist*) 0)))
F
CL-USER 14 > (f 9.0 3)
0 F > (9.0 3)
0 F < (6.0)
6.0
CL-USER 15 > (f -16.0 3)
0 F > (-16.0 3)
Break on entry to F
  1 (continue) return from break.
  2 (abort) return to level 0.
  3 return to top loop level 0.
  4 Destroy process.
Type :c followed by a number to proceed
```

4.4 Tracing methods

You can also trace methods (primary and auxiliary) within a generic function. The following example shows how to specify any qualifiers and specializers.

1. Type the following methods into the listener:

```
(defmethod foo (x)
 (print 'there))
(defmethod foo :before ((x integer))
 (print 'hello))
```

2. Next, trace only the second of these methods by typing the following definition spec.

(trace (method foo :before (integer)))

3. Test that the trace has worked by calling the methods in the listener:

```
CL-USER 226 > (foo 'x)

THERE

THERE

CL-USER 227 > (foo 4)

0 (METHOD FOO :BEFORE (INTEGER)) > (4)

HELLO

0 (METHOD FOO :BEFORE (INTEGER)) < (HELLO)

THERE

THERE

CL-USER 228 >
```

4.5 Trace variables

hcl:*max-trace-indent*

Variable

The maximum indentation used during output from trace.

hcl:*trace-indent-width*	Variable	
The additional amount by which tracing output is indented upor ing a deeper level of nesting.	n enter-	
hcl:*trace-level*	Variable	
The current depth of tracing.		
cl:*trace-output*	Variable	
The stream to which tracing sends its output by default.		
hcl:*traced-arglist*	Variable	
The variable that holds the arguments given to the traced function	on.	
hcl:*traced-results*	Variable	
The variable that holds the results from the traced function.		
The following four variables allow the output produced by tracing to be printed in a style that is controlled separately from normal printing:		
hcl:*trace-print-circle*	Variable	
The value to which *print-circle* is bound during output from trace.		
hcl:*trace-print-length*	Variable	
The value to which *print-length* is bound during output from trace.		
hcl:*trace-print-level*	Variable	
The value to which *print-level* is bound during output from trace.		

hcl:*trace-print-pretty*

Variable

The value to which *print-pretty* is bound during output from trace.

The Trace Facility

The Advice Facility

The advice facility provides a mechanism for altering the behavior of existing functions. As a simple application of this, you may supplement the original function definition by supplying additional actions to be performed before or after the function is called. Alternatively, you may replace the function with a new piece of code that has access to the original definition, but which is free to ignore it altogether and to process the arguments to the function and return the results from the function in any way you decide. The advice facility allows you to alter the behavior of functions in a very flexible manner, and may be used to engineer anything from a minor addition of a message, to a major modification of the interface to a function, to a complete change in the behavior of a function. This facility can be helpful when debugging, or when experimenting with new versions of functions, or when you wish to locally change some functionality without affecting the original definition.

Note: It can be very dangerous to put advice on functions defined by the system.Common LispWorks

Each change that is required should be specified using the defadvice macro. This defines a new body of code to be used when the function is called; this piece of code is called a piece of advice. Consider the following example:

In the above example you decided to print a message each time reverse is called. You called defadvice with a description of the function you wanted to alter, a name for the piece of advice, and the keyword :before to indicate that you want the code carried out before reverse is called. The rest of the call to defadvice specifies the additional behavior required, and consists of the lambda-list for the new piece of advice and its body (the lambda-list may specify keyword parameters and so forth). The advice facility arranges that print-advice is invoked whenever reverse is called, and that it receives the arguments to reverse, and that directly after this the original definition of reverse is called.

Pieces of advice may be given to be executed after the call by specifying **:after** instead of **:before** in the call to **defadvice**. So if you wished to add further code to be performed after **reverse** you could continue the session above as follows:

```
CL-USER 73 > (defadvice
                                (reverse after-advice :after)
                          (the-list)
                         (format t
                               "~%** After calling reverse on ~S **"
                               the-list))
NIL
CL-USER 74 > (reverse '("which" "way" "round"))
** Calling reverse on ("which" "way" "round") **
** After calling reverse on ("which" "way" "round") ** ("round"
"way" "which")
```

```
CL-USER 75 >
```

Note that after-advice also receives the arguments to reverse when it is called.

5.1 Combining the advice

We have already seen how a before and an after piece of advice may be combined, and this section describes the general algorithm. There are three types of advice: *before, after* and *around*. These resemble before, after and around methods in CLOS. There may be several pieces of each type of advice present for a particular function.

The first step in working out how the combination is done is to order the pieces of advice. All the around advice comes first, then all the before advice, then the original definition, and lastly the after advice. The order within each of the around, before and after sections defaults to the order in which the pieces of advice were defined (that is most recent first). See the description of defadvice in the *LispWorks Reference Manual* for details of how to control the ordering of advice within each section.

We shall now discuss what happens when a function that has advice is called. First we deal with the case when there is no around advice present. Here each of the pieces of before advice are called in turn, with the same arguments that were given to the function, next the original definition is called with these arguments, and finally each of the pieces of after advice is called in reverse order with the same arguments (so that by default the most recently added piece of after advice is invoked last). The results returned by the function call are the values produced by the last piece of after advice to be called (if there is one), or by the original definition (if there is no after advice).

Note that none of these bits of code should destructively modify the arguments that they receive. Adding a piece of before advice thus provides a simple way of specifying some additional action to be performed before the original definition, and before any older bits of before advice. Adding a piece of after advice allows you to specify extra actions to be performed after the original definition, and after any older bits of after advice. The advice facility automatically links together these bits of advice with the original function definition. Next we shall discuss the use of around advice, which provides you with greater control than do before and after advice. Let us suppose that a function that has some around advice is called. The arguments to the function are passed to the code associated with the first piece of around advice in the ordering, and the values returned by that piece of advice are the results of the function. There is no requirement for the advice to invoke any other pieces of advice, nor to call the original definition of the function.

However the code for any piece of around advice has access to the next member of the ordering, which it may invoke any number of times by calling callnext-advice. So it is possible for each piece of around advice to call its successor in the ordering if this is desired, and then the bits of around advice are called in turn in a similar fashion to our earlier description for before and after advice. However in the case of around advice the decision whether or not to call the next piece of advice is directly under your control, and you are free to modify the arguments received by the piece of advice, and to choose the arguments to be given to the next piece of advice if it is called.

If the last piece of around advice in the ordering calls call-next-advice, then it invokes the combination of before and after advice and the original definition that was discussed earlier. That is, the arguments to the call are given in the sequence described above to each of the before pieces of advice, then to the original definition and then to the after pieces of advice. The call to callnext-advice returns with the values produced by the last of these subsidiary calls, and the around advice may use these values in any way.

5.2 Removing advice

The macro delete-advice (or the function remove-advice) may be used to remove a named piece of advice. Since several pieces of advice may be attached to a single functional definition, the name must be supplied to indicate which one is to be removed.

```
CL-USER 40 > (delete-advice reverse after-advice)
NIL
CL-USER 41 > (delete-advice reverse print-advice)
NIL
```

5.3 Advice for macros and methods

As well as attaching advice to ordinary functions, it may also be attached to macros and methods.

In the case of a macro, advice is linked to the macro's expansion function, and so any before or after advice receives a copy of the arguments given to this expansion function (normally the macro call form and an environment). A simple example:

```
CL-USER 45 > (defmacro twice (b) `(+ ,b ,b))

TWICE

CL-USER 46 > (defadvice

(twice before-twice :before)

(call-form env)

(format t

"~%Twice with environment ~A and call-form ~A"

env call-form))

NIL

CL-USER 47 > (twice 3)

Twice with environment NIL and call-form (TWICE 3)

6
```

Note that the advice is invoked when the macro's expansion function is used. So if the macro is present within a function that is being compiled, then the advice is invoked during compilation of that function (and not when that function is finally used).

In the case of a method, the call to defadvice must also specify precisely to which method the advice belongs. A generic function may have several methods, so the call to defadvice includes a list of classes. This must correspond exactly to the parameter specializers of one of the methods for that generic function, and it is to that method that the advice is attached. For example:

```
CL-USER 45 > (progn
             (defclass animal ()
              (genus habitat description
              (food-type :accessor eats)
              (happiness :accessor how-happy)
              (eaten :accessor eaten :initform nil)))
             (defclass cat (animal)
              ((food-type :initform 'fish)))
             (defclass elephant (animal)
              (memory (food-type :initform 'hay)))
             (defmethod feed ((animal animal))
              (let ((food (eats animal)))
             (push food (eaten animal))
             (format t "~%Feeding ~A with ~A" animal
               food)))
            (defmethod feed ((animal cat))
             (let ((food (eats animal)))
            (push food (eaten animal))
            (push 'milk (eaten animal))
            (format t "~%Feeding cat ~A with ~A and ~A"
                 animal food 'milk)))
            (defvar *cat* (make-instance 'cat))
            (defvar *nellie* (make-instance 'elephant)))
*NELLIE*
CL-USER 46 > (feed *cat*)
Feeding cat #<CAT 6f35d4> with FISH and MILK
NIL
CL-USER 47 > (feed *nellie*)
Feeding #<ELEPHANT 71e7bc> with HAY
NIL
CL-USER 48 > (defadvice
             ((method feed (animal))
              after-feed :after)
             (animal)
             (format t "~%~A has eaten ~A"
                       animal (eaten animal)))
NIL
CL-USER 49 > (defadvice
             ((method feed (cat))
              before-feed :before)
             (animal)
             (format t "~%Stroking ~A" animal)
             (setf (how-happy animal) 'high))
```

NIL

```
CL-USER 50 > (feed *cat*)
Stroking #<CAT 6f35d4>
Feeding cat #<CAT 6f35d4> with FISH and MILK
NIL
CL-USER 51 > (feed *nellie*)
Feeding #<ELEPHANT 71eb7c> with HAY
#<ELEPHANT 71eb7c> has eaten (HAY HAY)
```

5.4 Example

So far you have only seen examples of before and after pieces of advice. This section contains some further examples. Suppose that you define a function alpha that squares a number, and then decide that you intended to return the reciprocal of the square instead. You might proceed as follows.

```
CL-USER 30 > (defun alpha (x) (* x x))
ALPHA
CL-USER 31 > (defadvice
(alpha reciprocal :around)
(num)
(/ (call-next-advice num)))
NIL
CL-USER 32 > (alpha -5)
1/25
```

First you change alpha to return the reciprocal of the square. Do this by defining an around method to take the reciprocal of the result produced by the next piece of advice (which initially is the original definition). Now suppose that you later decide that you would like alpha to return the sum of the squares of the reciprocals in a certain range. You can achieve this by adding an extra layer of around advice. This must iterate over the range required, summing the results obtained by the calls to the next piece of advice (which currently yields the reciprocal of the square of its argument).

```
CL-USER 36 > (defadvice
(alpha sum-over-range :around)
(start end)
(loop for i from start upto end
summing (call-next-advice i)))
NIL
CL-USER 37 > (alpha 2 5)
1669/3600
```

Note that alpha now behaves as a function requiring two arguments; the outer piece of around advice determines the external interface to the function, and uses the inner pieces of advice as it needs - in this case invoking the inner advice a variable number of times depending on the range specified. The use of the words "outer" and "inner" corresponds to earlier and later pieces of around advice in the ordering discussed above, but is more descriptive of their behavior.

You now realize that taking the reciprocal of zero gives an error. You decide that you wish to generate an error if alpha is called in such a way as to cause this, but that you want to generate the error yourself. You also decide to add a warning message for negative arguments. As you want these actions to be performed as the last (that is innermost) in the chain of around advice, you specify this in the call to defadvice by giving it a :where keyword with value :end.

```
CL-USER 41 > (defadvice
             (alpha zero-or-negative
               :around :where :end)
             (x)
             (unless (plusp x)
             (format t
            "~%**Warning: alpha is being called with ~A"
                x))
             (if (zerop x)
              (error "Alpha cannot be called with zero")
             (call-next-advice x)))
NIL
CL-USER 42 > (alpha -5 -2)
**Warning: alpha is being called with -5
**Warning: alpha is being called with -4
**Warning: alpha is being called with -3
**Warning: alpha is being called with -2
1669/3600
CL-USER 43 > (alpha 0 3)
**Warning: alpha is being called with 0
Error: alpha cannot be called with zero
 1 (abort) return to level 0.
 2 return to top loop level 0
Type :c followed by a number to proceed
CL-USER 44 : 1 > :a
```

Finally you decide to alter alpha yet again, this time to produce approximations to π . $\pi^2/6$ is the sum of the reciprocals of the squares of all the positive integers. So you can generate an approximation to π using the sum of the reciprocals of the squares of the integers from one to some limit. (In fact this is not an efficient way of calculating π , but it could be of interest.)

```
CL-USER 51 > (defadvice
                 (alpha pi-approximation :around)
                 (limit)
               (sqrt
                (* 6
                    (call-next-advice 1.0 limit))))
```

NIL

Next, try calling the falling in turn:

```
(alpha 10.0)
(alpha 100.0)
(alpha 1000.0)
pi
```

5.5 Advice functions and macros

The main functions used for advice are introduced below. See the *LispWorks Reference Manual* for full details.

The main macro used to define new pieces of advice is lw:defadvice.

Pieces of around advice should use the macro lw:call-next-advice to invoke the next piece of advice. As explained earlier this either calls the next piece of around advice (if one exists), or calls the combination of before advice, the original definition, and after advice. It may only be called from within the body of the around advice.

To remove a piece of advice, use the macro lw:delete-advice or the function lw:remove-advice.

Action Lists

Action-lists are a unified approach to various different mechanisms for running initializations, or "hook" functions at various points during the life of the system. They provide central gathering points for applications to trigger on system-wide events such as start-up, disk-save, and so on.

An action-list is a tagged list of data, to be executed (in some sense) in sequence whenever the circumstance identified by its tag occurs. It is expected that whatever code detects or causes the circumstance will take care of running the action-list.

An execution-function can be specified for the action-list when it is created. Otherwise, the default behavior is to treat the data of each action as a callable and apply it to any additional arguments specified at execution time. At its simplest, an action-list emulates (map nil 'funcall).

Names of action-lists and action-items are general lisp objects, compared with equalp. This allows strings and other objects to be used as unique identifiers.

Actions can be specified to depend on other actions; when defining an actionitem, you can say that it must be before or after other action-items using the :before and :after keywords. Aside from that, actions are assumed to have no dependencies, and no order of execution should be counted on for the actions in a list. You can (and are encouraged to) specify a documentation string for actionlists or action-items.

In addition you can create action-lists that are not registered globally. This allows applications to have disembodied action lists for their own internal purposes. The other action-list functions allow an action-list to be passed in instead of a name, to accommodate this.

6.1 Defining and undefining action lists

Action lists are defined using the define-action-list macro, and are undefined using the undefine-action-list. It is also possible to make unnamed, unregistered lists using make-unregistered-action-list.

define-action-list

Macro

```
define-action-list uid &key documentation sort-time dummy-actions default-order execution-function
```

The define-action-list macro defines an action list.

uid is a unique identifier, and must be a general lisp object, to be compared by equalp. It names the list in the global registry of lists. See makeunregistered-action-list to create unnamed, "unregistered" actionlists. The uid may be quoted, but is not required to be. It is possible, but not recommended, to define an action-list with unique identifier nil. If a registered action-list with the uid already exists (that is, one which returns t when compared with equalp), then notification and subsequent handling is controlled by the value of the *handle-existingaction-list* variable.

The *documentation* string allows you to provide documentation for the action list.

sort-time is a keyword specifying when added actions are sorted for the given list — either :execute or :define-action.

dummy-actions is a list of action-names that specify placeholding actions; they cannot be executed and are constrained to the order specified in this list, for example

'(:beginning :middle :end)

default-order specifies default ordering constraints for subsequently defined action-items where no explicit ordering constraints are specified. An example is

'(:after beginning :before :end)

execution-function specifies a user-defined function accepting arguments of the form:

(the-action-list other-args-list &rest keyword-value-pairs)

where the two required arguments are the action-list and a list of additional arguments passed to execute-actions, respectively. The remaining arguments are any number of keyword-value pairs that may be specified in the call to execute-actions. If no execution function is specified, then the default execution function will be used to execute the action-list.

undefine-action-list

Macro

undefine-action-list uid

The undefine-action-list flushes the specified list (and all its actionitems). If the action-list specified by *uid* does not exist, then handling is controlled by the value of the *handle-missing-action-list* variable.

When defining an action-list, the user may provide an associated executionfunction. When executing the action-list, this user-defined execution-function is used instead of the default execution-function, to map over and "execute" the action-list's action-items. The macrowith-action-list-mapping provides facilities to map over action-items (that is, their corresponding "data"). In addition, the with-action-list-error-handling macro provides a simple mechanism to trap errors and print warnings while executing each action-item.

All execution-functions are required to accept arguments of the form:

(action-list other-args &rest keyword-value-pairs)

where the two required arguments are the action-list and the list of additional arguments passed to execute-actions (see above), respectively. The remaining arguments are any number of keyword-value pairs that may be specified in

Action Lists

the call to execute-actions. See the LispWorks Reference Manual entries for with-action-list-mapping and with-action-item-error-handling for examples of execution-functions.

Actions are added to an action list using define-action, and are removed using undefine-action.

dotino	-action
ucinic	action

define-action nae-or-list action-name data &rest specs

This macro adds a new action to the list specified by *name-or-list*, which will be executed according to the action list's execution function.

Macro

Macro

Variable

Variable

undefine-action

undefine-action nae-or-list action-name

This macro removes the action specified by *action-name* from the list specified by *name-or-list*.

6.2 Exception handling variables

The following global variables are used to control the handling of exceptions:

handle-existing-action-list

A list containing either :warn or :silent, determining whether to notify the user, and either :skip or :redefine to determine what to do about an action-list operation when the action-list already exists. The default value is '(:warn :skip). It is used by the define-action-list macro.

handle-existing-action-in-action-list

A list containing one of :warn, or :silent, determining whether to notify the user, and one of :skip, or :redefine, to determine what to do about an action definition when the action already exists in the given action-list. The default value is '(:warn :skip). It is used by defineaction.

54

handle-missing-action-list

A keyword; one of :warn, :error, or :ignore, denoting how to handle an operation on a missing action-list. The default value is :error. It is used by undefine-action-list, print-actions, execute-actions, define-action and undefine-action.

handle-missing-action-in-action-list

A keyword; one of :warn, :error or :ignore, denoting how to handle an operation on a missing action. Its default value is :warn. It is used by undefine-action.

6.3 Other variables

default-action-list-sort-time

Contains a keyword that is either :execute or :define-action, denoting when actions in action-lists are sorted (see define-action-list for an explanation of ordering specifiers). Actions are sorted either at time of definition (:define-action) or when their action-list is executed (:execute). The default sort time is :execute.

6.4 Diagnostic utilities

Two diagnostic functions are provided: print-actions which prints out the actions on an action list. and print-action-lists, which provides a list of all the defined action lists.

print-actions

print-actions name-or-list &optional stream

Generates a listing of the action items on this action-list, in order. If the action-list specified by name-or-list does not exist, then this is handled according to the value of *handle-missing-action-list*.

```
Variable
```

Variable

Variable

Function

print-action-lists

print-action-lists & optional stream

Generates a listing of all the action lists in the global registry. (The ordering of the action lists here is essentially random.)

6.5 Examples

This example illustrates "typical" use of action lists. The define-action forms might be scattered across several files (mail-utilities.lisp, caffeine.lisp, and so on). Each of the functions, such as read-mail, dont-panic, and so on, take one argument: hassled-p.

```
(in-package "CL-USER")
(define-action-list "On arrival at office"
    :documentation "Things to do in the morning"
    :dummy-actions '("Look busy")
    :default-order '(:before "Look busy"))
(define-action "On arrival at office" "Read mail" 'read-mail)
(define-action "On arrival at office" "Greet co-workers"
                 'say-hello)
(define-action "On arrival at office" "Drink much coffee"
                 'wake-up:after "Locate coffee machine")
(define-action "On arrival at office" "Locate coffee machine"
                 'dont-panic)
(defun my-morning (hassled-p Monday-p)
   (execute-actions ("On arrival at office"
                     :ignore-errors-p Monday-p)
                 hassled-p)
    <rest of my-moring code goes here>)
```

This example illustrates use of execution-functions and post-processing

(in-package "CL-USER")

Here are the implementation details, which are hidden from the "user".

(defstruct (thing (:constructor make-thing (name number)))
 name
 number)

The interface is given below. The internals of the mapping mechanism are hidden.

Action Lists

7

The Compiler

The compiler translates Lisp forms and source files into binary code for the host machine. A compiled Lisp function, for instance, is a sequence of machine instructions that directly execute the actions the evaluator would perform in interpreting an application of the original source lambda expression. Where possible the behaviors of compiled and interpreted versions of the same Lisp function are identical. Unfortunately the definition of the Common Lisp language results in certain unavoidable exceptions to this rule. The compiler, for instance, must macroexpand the source before translating it; any side effects of macro-expansion happen only once, at compile time.

By using declarations, you can advise the compiler of the types of variables local to a function or shared across an application. For example, numeric operations on a variable declared as a single-float can be compiled as direct floating-point operations, without the need to check the type at execution time. You can also control the relative emphasis the compiler places on efficiency (speed and space), safety (type checking) and support for debugging. By default the compiler produces code that performs all the necessary type checking and includes code to recover from errors. It is especially important that the type declarations be correct when compiling with a safety level less than 3 (see later in this chapter for more details).

When compiling a Lisp source file, the compiler produces its output in a format that is much faster to load than textual Lisp source — the "fasl" or "fastload" form. Fasl files contain arbitrary Common Lisp objects in a pre-digested form. They are loaded without needing to use the expensive read function. A series of "fasl-loader" routines built into LispWorks interpret the contents of fasl files, building the appropriate objects and structures in such a way that objects that were eq before fasl-dumping are created eq when fasl-loaded.

Fasl files are given pathname extensions that reflect the target processor they were compiled for; as the fasl files contain processor specific instruction sequences it is essential that the loader be able to distinguish between files compiled for different targets. These pathname extensions always end in "fasl". See dump-forms-to-file in the *LispWorks Reference Manual* for details of all the possible fasl file extensions.

7.1 Compiling a function

The function compile takes a symbol as its first argument, and an interpreted function definition (a lambda expression) as its second, optional, argument. It compiles the definition and installs the resultant code as the symbol-function of the symbol (unless the symbol was nil). If the definition is omitted then the current symbol-function of the symbol is used. Below are some examples:

```
CL-USER 3 > (compile (defun fred (a b)
(dotimes (n a) (funcall b))))
FRED
FRED
NIL
CL-USER 4 > (funcall (compile nil '(lambda (n)
(* n n))) 7)
; NIL
49
CL-USER 5 > (compile 'ident-fun '(lambda (x) x))
;IDENT-FUN
IDENT-FUN
NIL
NIL
```

7.2 Compiling a source file

The function compile-file takes a pathname as its argument and compiles all the forms in the file, producing a corresponding fasl file (with pathname derived from the source pathname). Any side effects in the source file are only felt once the compiled file is subsequently loaded. Many proclamations, for example, are not visible at compile time. The eval-when special form can be used to force such side effects to take effect at the time of compilation, rather than loading.

7.3 How the compiler works

Conceptually the compiler can be viewed as performing a series of separate passes.

- In the first pass the source code is macro expanded in the appropriate macro environment.
- A series of source to source optimizing transformations are performed to simplify the source tree. Type declarations are used to select specialized, efficient versions of low level functions.
- A graph is generated from the source tree. The structure of the graph reflects the flow of control in the tree. The nodes of the graph contain blocks of intermediate code for an abstract machine with byte addressing and an infinite set of registers. Register allocation is performed based on data flow analysis and machine specific rules concerning live ranges across code fragments.
- The blocks of intermediate code are translated into a single linear sequence of target machine code though a process of template matching.
- Finally the relative branch instructions are "fixed up" to point to the correct locations in the code sequence.

The compiler is in fact much more complex than this model might suggest. Machine specific optimizations, for example, can be included in any of the passes. The distinction between passes is also not as simple as that listed above. However, this description is sufficient to allow the programmer to make optimal use of the compiler.

7.4 Compiler control

There are ways to control the nature of compiled code via the declare special form and proclaim function. See later in this chapter for fuller discussion of these two forms.

In particular there are a set of optimize qualities which take integral values from 0 to 3, in order to control the trade-offs between code size, speed, compilation time, debuggability of the resulting code, and the safety of the code (whether type checks are omitted). For example:

```
(proclaim '(optimize (speed 3) (safety 0) (debug 0)))
```

tells the compiler to concentrate on code speed rather than anything else, and

```
(proclaim '(optimize (safety 3)))
```

ensures that the compiler never takes liberties with Lisp semantics and produces code that checks for every kind of signallable error.

The important declarations to the compiler are type declarations and optimize declarations. To declare that the type of the value of a variable can be relied upon to be unchanging (and hence allow the compiler to omit various checks in the code), say:

```
(declare (type the-type variable * )
```

Optimize declarations have various qualities, and these take values from 0 to 3. The keywords are safety, fixnum-safety, sys:interruptable, debug, speed, Compilation-speed, and space.

Most of the qualities default to 1 (but safety and fixnum-safety default to 3 and interruptable defaults to 0). You can either associate an optimize quality with a new value (with local lexical scope if in declare, and global scope if proclaim), or just give it by itself, which implies the value 3 (taken to mean "maximum" in some loose sense).

Thus you ensure code is at maximum safety by:

```
(proclaim '(optimize (safety 3)))
```

or

```
(proclaim '(optimize safety))
```

and reduce debugging information to a minimum by

(proclaim '(optimize (debug 0)))

Normally code is interruptible, but when going for the extreme levels of speed and "undebuggability" this ceases to be the case unless you also ensure it thus:

```
(proclaim '(optimize (debug 0) (safety 0) (speed 3)
interruptable))
```

The levels of safety have the following implications:

- 0 implies no type checking upon reading or writing from defstructs, arrays and objects in general, nor any range checking of subscripts.
- 1 implies no type checking upon reading from defstructs, arrays and objects in general, nor any range checking of subscripts when reading.
- 2 implies generally safe code, but allows type and fixnum-safety declarations to take effect.
- 3 (default) implies complete type and range checking, and disallows fixnum-safety and type declarations from taking any effect.

The levels of fixnum-safety have the following implications:

- 0 implies no type checking of arguments to numeric operations, which are assumed to be fixnums. Also the result is assumed, without checking, to not overflow this level means single machine instructions can be generated for most common integer operations, but risks generating values that may confuse the garbage collector.
- 1 implies that numeric operations do not check their argument types (assumed fixnum), but do signal an error if the result would have been out of range.
- 2 implies that numeric operations signal an error if their arguments are non-fixnum, and also check for overflow.
- 3 (default) implies complete conformance to the semantics of Common Lisp numbers, so that types other than integers are handled in compiled code.

The Compiler

The effects of combining these qualities is summarized below:

Keyword settings	Operations
safety=0	Array access optimizations
debug>0	Dumps symbol names for arglist
debug>=2	"Uniquely spills" various registers (some- times only when source debugging is on)
debug<1	Does not generate any debug info at all
debug=3	Avoids make-instance and find-class optimizations
debug=3	Avoids gethash and puthash optimiza- tions
debug=3	Avoids 1db and dqb optimizations
debug=3	Avoids an optimization to last
safety>1	Be careful when multiple value counts are wrong
safety<1	Do not check array indices during write
safety<2	Do not check array indices during read
debug<3	Eliminate tail recursion
speed>space	Inline map functions (unless debug>2)
debug<=2	"Tail merging"
debug<2 and safety<2	"Self calls"
safety>=2	"Check get special"
safety<1	Do not check array indices during write
safety<2	Do not check array indices during read

Table 7.1 Combining debug and safety levels in the compiler

Keyword settings	Operations
safety>=1	"Check structure access"
debug>=1	Call count count
safety>1	Check number of args
<pre>safety>=1 Or interruptible>0</pre>	Check stack overflow
safety>1	Ensures the thing being funcalled is a function
safety<3 and fixnum-safety=2	Fixnum-only arithmetic except where overridden by type declarations
safety<3 and fixnum-safety=1	No fixnum overflow checks
safety<3 and fixnum-safety=0	No fixnum arithmetic checks at all
safety>2	char= checks for arguments of type char- acter
safety>=2	Ensures symbols in progv
debug=3	Avoids "ad hoc" predicate type transforms
compilation-speed=3	Reuse virtual registers in very large func- tions

Table 7.1 Combining debug and safety levels in the compiler

The other optimize qualities are: speed — the attention to fast code, space — the degree of compactness, compilation-speed — speed of compilation, interruptable — whether code must be interruptible when unsafe.

Note that if you compile code with a low level of safety, you may get segmentation violations if the code is incorrect (for example, if type checking is turned off and you supply incorrect types). You can check this by interpreting the code rather than compiling it.

7.5 Declare, proclaim, and declaim

declare

Special form

declare (declaration *)

There are two distinct uses of declare, one is to declare Lisp variables as "special" (this affects the semantics of the appropriate bindings of the variables), and the other is to provide advice to help the Common Lisp system (in reality the compiler) run your Lisp code faster, or with more sophisticated debugging options.

The special form declare behaves computationally as if it is not present (other than to affect the semantics), and is only allowed in certain contexts, such as after the variable list in a let, do, defun, etc.

(Consult the syntax definition of each special form to see if it takes declare forms and/or documentation strings.)

See the *LispWorks Reference Manual* for more detailed information on declare, including some LispWorks extensions to Common Lisp.

proclaim

Function

proclaim declaration-list

declaration-list must be a list of declaration forms to be put into immediate and pervasive effect.

Unlike declare, proclaim is a function that parses the declarations in the list (usually a quoted list, note), and puts their semantics and advice into global effect. This can be useful when compiling a file for speedy execution, since a proclamation such as:

```
(proclaim '(optimize (speed 3) (space 0) (debug 0)))
```

means that the rest of the file is compiled with these optimization levels in effect. (The other way of doing this is to make appropriate declarations in every function in the file).

proclaim simply returns nil.

declaim

Macro

This is a macro equivalent to proclaim.

Below are some examples:

```
(proclaim '(special *fred*))
(proclaim '(type single-float x y z))
(proclaim '(optimize (safety 0) (speed 3)))
```

As proclaim involves parsing a list of lists of symbols and is intended to be used a few times per file, its implementation is not optimized for speed - it makes little sense to use it other than at top level.

Do not forget to quote the argument list if it is a constant list. (proclaim (special x)) attempts to call function special.

7.5.1 Naming conventions

Exercise caution if you declare or proclaim variables to be special without regard to the naming convention that surrounds their names with asterisks.

7.6 Compiler parameters affecting LispWorks

There are six compiler parameters that control the generation of information used by various LispWorks utilities, such as the debugger, and also by various editor commands, such as **Show Paths From**. By default, these parameters are all t, which allows you to use all the features of these utilities, at the expense of increasing compilation times.

These variables are initially set to t (in the configuration file configure.lisp). To speed up compilation times, you should set these variables to nil. The variables can be controlled as a group by using the "master-toggle" function lispworks:toggle-source-debugging. See under this function in the *LispWorks Reference Manual* for details of all the variables that it controls. The Compiler

Storage Management

This chapter introduces some basic ideas of storage management, and then discusses the LispWorks storage management system in more detail. The chapter also introduces the functions and macros needed to control storage management. Full details of all the symbols mentioned here are given in the *LispWorks Reference Manual*.

8.1 Introduction

Automatic memory management is one of the most significant features of a Lisp system. Whenever an object, such as a cons cell, is required to hold an aggregate of values, the system calls the appropriate function to create a new object and fill it with the intended values. The programmer need not be concerned with the low level allocation and management of memory as the Lisp system provides this functionality automatically.

When an object is no longer required (that is, it has become "garbage"), the system must automatically reclaim ("collect") the space it occupies and reallocate the space to a new object. Whenever the space for new objects is exhausted, a "garbage collector" is run to determine (by a process of elimination) all the existing objects that are still required by the running program. Any other objects still in the image are necessarily garbage, and the space they occupy can be reclaimed. Garbage collection with a naive algorithm is extremely inefficient. The time required to scan an entire image, which may occupy many megabytes of memory, is prohibitive; especially when the collector must perform the scan in a small, fixed, workspace. The remainder of this chapter describes the Lisp-Works garbage collector in more detail.

8.2 Generations and segments

The LispWorks garbage collector works in unison with the storage allocator to arrange allocated objects in a series of "generations". Each generation contains objects of a particular age. In practice, most Lisp data objects are only required for a very short period of time i.e. are ephemeral. The LispWorks garbage collector concentrates its efforts on repeatedly scanning the most recent generation. Such a scan requires only a fraction of a second and reclaims most of the space allocated since the last collection. Any object in the most recent generation that survives a number of such collections is promoted to the next youngest generation. Eventually this older generation becomes full, and only then is it collected.

In memory, a generation consists of a chain of segments. Each segment is a contiguous block of memory, beginning with a header and followed by the allocation area. The generations are numbered from 0 upwards, so that generation 0 is the youngest.

The first generation normally consists of two segments: the first segment is relatively small, and is where most of the allocation takes place. The second segment is called the big-chunk area, and is used for allocating large objects and when overflow occurs (see below for a discussion of overflow).

The second generation (generation 1) is an intermediate generation, for objects that have been promoted from generation 0 (typically for objects that live for some minutes). Long-lived objects created dynamically are normally allocated in the next generation, generation 2.

Note that the division between the generations is a result of the promotion mechanism, and is not a property of a piece of code itself. A piece of system software code that is loaded in the system (for example, a patch) is treated the same as any other code. The garbage collection code is explicitly loaded in the static area using the function switch-static-allocation.

8.3 Allocation of objects to generations

Normal allocation is done from a buffer, called the small objects buffer. The Garbage Collector (GC) maintains a pointer to the beginning and end of the buffer, and allocates from it by moving one of the boundaries. When the buffer becomes too small the GC finds another free block and makes that the buffer.

The minimum and maximum size of free block that the GC uses for the small objects buffer can be set by set-gc-parameters, using the keywords :minimum-buffer-size and :maximum-buffer-size. If the minimum size is too small, the system allocates buffers more frequently, thus slowing the program. Making the minimum too big causes more fragmentation, because small free blocks are not used. There is no easy way to determine the optimal values for the small objects buffer, except by experiment.

When there is an overflow the small object buffer is allocated in the big-chunk area, and then a bigger buffer is allocated (see below).

8.4 Allocation of static objects

Objects that cannot be moved are allocated in special segments, called static segments. These can be in any generation, but are in generation 2 by default.

Such objects include:

- Code that must not move during garbage collection, in particular the code and data of the garbage collector itself
- Objects allocated explicitly in the static area, by in-static-area or by use of switch-static-allocation.

Because static objects are not allowed to move, the static segments are not allowed to move. This implies that if there is a static segment in a high address the image size cannot be reduced below this size. Applications that use a lot of static area normally allocate additional static segments, and thus grow without being able to shrink again. This can be prevented by enlarging the initial static segment, which is in a low address. Use the function <code>lw:enlargestatic</code> to increase the size of the initial static segment. (Use (room t) to find its current size.)

8.5 Allocation in different generations

Objects that are known to have long life can be allocated directly in a higher generation, by using allocation-in-gen-num and set-default-generation. Note that both these functions have a global effect, i.e. any object allocated after a call to set-default-generation or within the body of allocation-ingen-num is allocated in the specified generation, unless it is explicitly allocated in a different generation. Therefore careless use of these functions may lead to allocation of ephemeral garbage in high generations, which is very inefficient. Conversely, if a long-lasting object is allocated to a low generation, it has to survive several garbage collections before being automatically promoted to the next generation.

Interned symbols (and their symbol names) are treated in a special way, because they are assumed to have a long life. They are allocated in the generation specified by lispworks:*symbol-alloc-gen-num*, which defaults to 2.

8.6 Mark and sweep

Mark and sweep is the basic operation of reclaiming memory, and it is done in two stages:

Mark	All objects that are alive in the generation being gar- bage collected and in younger generations are marked as alive. (Alive means pointed to by some other live object.)
Sweep	All unmarked objects in the generations being garbage collected are added to the free blocks, and all marked objects are unmarked.

A mark and sweep operation is always on all the generations from 0 to a specific number.

A mark and sweep operation can be caused explicitly by calling mark-andsweep.

8.7 Promotion

Promotion is the process of moving objects from one generation to the next generation. An object is marked for promotion after surviving a specific number of mark and sweep operations, but may be promoted before that. The number of survivals is specific to each segment, and can be set by set-promotion-count.

8.8 Garbage collection strategy

When the Garbage Collector runs out of memory, it has to find more memory. Normally (that is, when allocating in generation 0) the first operation is a mark and sweep. Before performing the mark and sweep, the GC compares the amount of memory allocated since the previous mark and sweep with the <code>:minimum-for-sweep</code> value, which is set by <code>set-gc-parameters</code>. If the amount allocated is less than <code>:minimum-for-sweep</code> the GC does not do a mark and sweep, but causes an overflow (described below). This prevents an excessive number of mark and sweep operations in periods when the program allocates a large amount of data which stays alive.

Note that the GC monitor window does not indicate a mark-and-sweep of generation 0, as this operation takes a small amount of time (To change the display would take longer than the mark-and-sweep operation itself.)

If more than :minimum-for-sweep has been allocated, a mark and sweep operation takes place. After this operation the GC checks that the segment it was trying to allocate to has more free space than the minimum free space for this segment. If the remaining free space is less than minimum-free-space, the GC tries to create more free space by promoting objects from the segment.

Before promoting, the GC performs two checks. First, it checks that there are enough objects marked for promotion to justify a promotion operation. The minimum free space for a segment is set by set-minimum-free-space, and can be shown by (room t).

Second, the GC checks that there is enough free space in the next generation to accommodate the promoted objects. If there is insufficient space, the GC tries to free some, either by a mark and sweep on the next generation, promoting the next generation, or by enlarging the generation.

The minimum amount of space for promotion is the value minimum-for-promote, which is set by set-gc-parameters.

If there is insufficient space, and there are not enough objects marked for promotion, the GC increases the size of the image, by overflow, as described below.

8.9 Overflow

If the amount allocated from the previous mark and sweep operation is less than :minimum-for-sweep, the GC does not perform a mark and sweep. Instead it allocates a small-objects buffer in the big-chunk area (the second segment in the first generation). The minimum and maximum sizes of this buffer are specified by :minimum-overflow and :maximum-overflow, which can be set by set-gc-parameters. If the GC fails to find a buffer of this size, it looks for a smaller buffer, and if that fails it enlarges the big-chunk area (and the process size) by the amount needed to allocate a buffer of the size of the currently allocated area in generation 0, up to a maximum amount specified by :maximum-overflow.

8.10 Behavior of generation 1

When objects are promoted from generation 0 to 1, and there is not enough space in generation 1, the GC tries to free space in generation 1. The first step is to check if sufficient space can be freed by promoting the objects marked for promotion. If this is the case the GC promotes these objects from generation 1 to generation 2. (In practice, this rarely happens.) If this check fails the GC marks and sweeps generation 1. If not enough space is freed by this mark-and-sweep, than either all the objects in generation 1 are promoted, or generation 1 is expanded. This is controlled by <code>lispworks:expand-generation-1</code>, which specifies whether expansion or promotion takes place.

If generation 1 is expanded, the amount it tries to expand by is the value :new-generation-size (set by set-gc-parameters) in words (i.e. multiples of 4 bytes), or the amount of free space needed, whichever is bigger. If :new-generation-size is 0, it is not expanded. In this case part of the objects marked for promotion are not promoted.

8.11 Behavior of generation 2

Normally generation 2 is not garbage collected. If the GC runs out of space in this generation, it expands it, using the value of :new-generation-size multiplied by two. Garbage collection of generation 2 can be caused by calling the function collect-generation-2 with appropriate argument.

8.12 Timing the garbage collector

The macro extended-time is useful when timing the garbage collector.

8.13 Summary of garbage collection symbols

The remainder of this chapter summarizes which functions are useful in which circumstances. For full details of these functions, see the *LispWorks Reference Manual*.

8.13.1 Determining storage usage

To determine storage usage (useful when benchmarking), use the functions room, total-allocation and find-object-size.

8.13.2 Allocating in specific generations

To control the allocation of objects to generations, use allocation-in-gennum, get-default-generation, set-default-generation and *symbolalloc-gen-num*.

8.13.3 Controlling a specific generation

To control the behavior of a specific generation, use clean-generation-0, collect-generation-2, collect-highest-generation, expand-generation-1, set-minimum-free-space and set-promotion-count.

8.13.4 Reducing image size

To reduce the size of the whole image, use clean-down.

8.13.5 Controlling the garbage collector

To control the actions of the garbage collector, use the symbols avoid-gc, get-gc-parameters, gc-if-needed, mark-and-sweep, normal-gc, set-gc-parameters, without-interrupts and with-heavy-allocation.

8.13.6 Using special actions

In addition, users may want to perform special actions when certain types of object are garbage collected, using the functions add-special-free-action, flag-special-free-action, flag-not-special-free-action and remove-special-free-action.

For example, when a file stream is closed, and so garbage collected, the file descriptor must be closed also. This operation can be performed as a special action.

The Profiler

The LispWorks profiler provides a way of empirically monitoring execution characteristics of Lisp programs. The data obtained can help to improve the efficiency of a Lisp program by highlighting those procedures which are commonly used or particularly slow, and which would therefore benefit from optimization effort.

For complete details of the functions and symbols introduced in this chapter, see the *LispWorks Reference Manual*.

9.1 What the profiler does

With the profiler running, the Lisp process is interrupted regularly at a specified time interval until the profiler is turned off. Having halted the execution of the process the profiler scans the execution stack and records the name of every function found. A special note is made of which function is at the top of the stack. When profiling stops the profiler presents aggregated information about each function as follows:

- The number of times the function was called.
- The number of times the function was found on the stack by the profiler, both in absolute terms and as a percentage of the total number of scans of the stack.

• The number of times the function was found on the top of the stack, both in absolute terms and as a percentage of the total number of scans of the stack.

9.2 Setting up the profiler

Before a profiling session can start several parameters must be set, using the function set-up-profiler. This function is introduced here and the full syntax is given in the *LispWorks Reference Manual*. There are three main areas to consider: the symbols to be profiled, the time interval between samples, and the kind of profiling required.

- It is possible to keep track of every function called during a particular computation, but significant effort is involved in determining which symbols are suitable for profiling and in keeping track of the results. To minimize this effort you need to specify which symbols to profile, either by naming the required symbols, or by naming a package, all of whose symbols are profiled. The profiler first checks that these symbols have indeed got function definitions and are therefore suitable for profiling.
- You might want to specify the time interval between interrupts. The resolution of this value is clearly dependent on the operating system. For example, the greatest number of interrupts that can be sent to a process under SunOS is 100 per second. This number is important, because with these statistical methods of program profiling the accuracy of the results increases with the number of samples taken. In most cases the default values, 10 ms, is adequate.

Below is an example of setting up the profiler:

```
(set-up-profiler :symbols '(car cdr))
```

Here the functions car and cdr are going to be profiled.

The function set-up-profiler adds symbols to the *profile-symbol-list*. The functions add-symbol-profiler and remove-symbol-profiler can also be used to change the symbols profiled.

The function set-profiler-threshold can be used with reset-profiler to control the effects of repeated profiler runs.

9.3 Running the profiler

To profile a Lisp form using the set-up from the previous example, type:

(profile <forms>)

A typical output would be:

Profile sta	cks called 40 tim	es		
Symbol	Called	Profile (%)	Top	(%)
CAR	400	20 (50)	2	(5)
CDR	300	15 (39)	3	(7)
Top symbol	not monitored 88%	of the time.		

This means that during the execution of the form, the function car was called four hundred times, the function cdr was called three hundred times, and the Lisp process was interrupted forty times by the profiler. In half of these interrupts it found the function car on the stack, but only on two of these occasions was car on the top of the stack. Thirty-five times the function on the top of the stack was neither car nor cdr.

You can control the order of the output from the profiler using print-profile-list.

9.4 Interpretation of profiling results

The most important figures are the number of times a function was called along with the amount of time it was found on top of the stack. Just because a function is found on the stack does not mean that it using up much processing time, but if it is found consistently on the top of the stack then it is likely that this function has a significant execution time. Similarly functions that are called most often are likely to have the most significant effect on the program as a whole.

It must be remembered that the numbers produced are from random samples and thus it is important to be careful in interpreting their meaning. The rate of sampling is always coarse in comparison to the function call rate and so it is possible for strange effects to occur and significant events to be missed. For example, "resonance" may occur when an event always occurs between regular sampling times, though in practice this does not appear to be a problem.

9.5 Profiling pitfalls

Profiling should only be attempted on compiled code. If it is done on interpreted code, the interpreter itself is profiled, and this upsets the results for the actual Lisp program.

Macros cannot be profiled as they are expanded during the compilation process. Similarly some Common Lisp functions may be present in the source code but not in the compiled code as they are transformed by the compiler. For example:

```
(member 'x '(x y z) :test #'eq)
```

is transformed to:

(memq 'x '(x y z))

by the compiler and therefore the function member is never called.

Recursive functions need special attention. A recursive function may well be found on the stack in more than one place during one interrupt. The profiler counts each occurrence of the function. Hence the total number of times a function is found on the stack may be much greater than the number of times the stack is examined.

Care must be taken when profiling structure accessors. Structure accessors compile down into a call to a closure of which there is one for all structure setters and one for all structure getters. Therefore it is not possible to profile individual structure setters or getters by name.

It must be remembered that even though a function is found on the stack this does not mean that it is active or that it is contributing significantly to the execution time. However the function found on the top of the stack is by definition active, and thus this is the more important value.

It is quite possible that the amount of time the top symbol is monitored is significantly less than 100% despite the profiler being set to profile all the known functions of the application. This is because at the time of the interrupt an internal system function may well be on the top of the stack.

It is possible to profile all the symbols in the system by setting up the profiler as follows:

```
(set-up-profiler :package (list-all-packages))
```

9.6 Profiling and garbage collection

The macro extended-time provides useful information on garbage collection activities. See the *LispWorks Reference Manual* for full details.

The Profiler

10

Simple Customization of LispWorks

This chapter gives examples of how to make minor changes to the LispWorks interface.

There are a number of files that contain configuration and initialization information:

- The LispWorks file config/configure.lisp contains many default configuration settings. You should customize this file when you install LispWorks.
- The LispWorks file config\key-bindings.lisp gives the default editor key bindings.
- Your home directory may contain a .lispworks file containing Lisp forms to be evaluated on startup.

By default, your home directory is located in C:\users\, and has the same name as your Windows user name. Thus, if you log into Windows with the name john, your home directory is C:\users\john\.

Many of the customizations discussed in this chapter involve making changes to your .lispworks file.

10.1 Customizing the editor

This section explains some of the customizations you can make to the Common LispWorks editor tool.

10.1.1 Placing definitions at the top of a window

When using the editor, **Find Dspec** and similar operations retrieve the file containing a definition and place it in a buffer with the relevant definition visible. By default, the start of the definition is in the middle of the Editor window. The following example shows how to make the definition appear at the top of the window

```
(setq editor:*tags-definition-top-window* t)
```

This variable is specified in the file configure.lisp.

10.1.2 Specifying the number of editor windows

You can specify the maximum number of editor windows that are present at any one time. For example, to set the maximum to 1:

```
(setq editor:*maximum-ordinary-windows* 1)
```

This variable is specified in the configure.lisp file.

10.1.3 Binding commands to keystrokes

You can bind existing editor commands to different keystrokes, using editor:bind-key. Supplied with LispWorks is the file configkey-binds.lisp, which gives sets up the standard key bindings for LispWorks.

The following example shows how to rebind ? so that it behaves as an ordinary character in the echo area of a Common LispWorks tool — this can be useful if your symbol names include question marks.

```
(editor:bind-key "Self Insert" #\? :mode "Echo Area")
```

Since ? is then no longer available for help, you may wish to rebind help to Ctrl+?.

```
(editor:bind-key "Help on Parse" #\C-? :mode "Echo Area")
```

10.2 Using ! for :redo

The default way of redoing the previous command from the command history is via :redo. If you want to use ! (exclamation mark) instead of :redo, add the following to your .lispworks file:

```
(set-macro-character #\!
  #'(lambda (stream char)
  ':redo))
```

You may wish during some sessions to reset ! back to its normal role as a character. Type in:

```
(set-syntax-from-char #\@ #\!)
```

10.3 Customizing LispWorks for use with your own code

This section contains some information on customizations you can make in order to make developing your own code a little easier.

10.3.1 Preloading selected modules

If you frequently use some code that is normally supplied as separate modules, you can load them at start-up time from your initialization file. This file is called .lispworks by default, but can be changed to be any other filename by using the global preferences dialog. See the *Common LispWorks User Guide* for more detail.

For example, to load the dynamic-completion code every time you start LispWorks, include the following in your initialization file.

```
(require "dynamic-complete")
```

10.3.2 Creating packages

When writing your own code that uses, for instance, the capi package, create a package of your own that uses capi — do not work directly in the capi package. By doing this you can avoid unexpected name clashes. Simple Customization of LispWorks

11

Multiprocessing

LispWorks supports "lightweight" processes. The programming environment, for example, makes extensive use of this mechanism to create separate processes for the various tools.

This chapter covers the following topics:

- Introduction to processes
- The process programming interface
- Locking

11.1 Introduction to processes

A process can be in one of three different states: *running*, *waiting*, and *inactive*. When a process is *waiting*, it is still active, but is waiting for the system to wake it up and allow its computation to restart. A process that is *inactive* has stopped, because it has an arrest "reason".

For a process to be active (that is, running or waiting), it must have at least one run reason and no arrest reasons. If, for example, it was necessary to temporarily stop a process, it could temporarily be given an arrest reason. However the arrest reason mechanism is not commonly used in this manner. The process that is currently executing is termed "the current process" and is the current value of the variable mp:*current-process*. The current process continues to be executed until either it becomes a waiting process, by calling mp:process-wait Or mp:process-wait-with-timeout, or it allows itself to be interrupted, by calling mp:process-allow-scheduling (or its current timeslice expires and it involuntarily relinquishes control).

The system runs the waiting process with the highest priority. If processes have the same priority then the system treats them equally and fairly. This is called round robin scheduling.

The simplest way to create a process is to use mp:process-run-function. This creates a process with the specified name which commences by applying the specified function to arguments. mp:process-run-function returns immediately and the newly created process runs concurrently.

11.2 The process programming interface

11.2.1 Creating a process

To create a new process, use mp:process-run-function. This in turn calls mp:create-process.

11.2.2 Finding out about processes

The system initializes a number of processes on startup. These processes are specified by mp:*initial-processes*.

The current process is specified by mp:*current-process*. A list of all the current processes is returned by mp:list-all-processes. The function mp:ps is analogous to the UNIX command ps, and returns a list of the processes in the system, ordered by priority.

To find a process when you know its name, use mp:find-process-from-name. To find the name, when you have the process, use mp:process-name. The variable mp:*process-initial-bindings* specifies the variables that are initially bound in a process.

When a process has stopped, you can find a list of reasons why by calling mp:process-arrest-reasons. To obtain a list of the reasons why a process is

running, call mp:process-run-reasons. Both these lists can be changed using setf, though it is not normally necessary to add arrest reasons. To find the priority of a process, use mp:process-priority. This can be changed using mp:change-process-priority.

11.2.3 Interrupting a process

To interrupt a running process, use mp:process-interrupt, Or mp:processkill. To break a process and enter the debugger, use mp:process-break.

To suspend a process until a predicate is t, use mp:process-wait or mp:process-wait-with-timeout. The function mp:process-wait-function returns a function that specifies a reason for the process waiting.

To control whether or not a process can be interrupted, use mp:without-preemption Or mp:without-interrupts.

11.2.4 Multiprocessing

To initialize multiprocessing, use mp:initialize-multiprocessing. It is not necessary to use this function when the LispWorks environment is already running. This function does not return until multiprocessing has terminated.

You can call mp:initialize-multiprocessing from the TTY interface, which generates a default Listener process if no other processes are specified by mp:*initial-processes*

Note: You cannot currently save an image with multiprocessing running.

11.2.5 Example

The following example allows two (or more) multiplication tables to be printed out simultaneously.

First, the function to print out a multiplication table.

```
(in-package "USER")
(defun print-table (number total stream)
(do ((i 1 (+ i 1)))
((> i total))
(format stream "~S X ~S = ~S~%" number i (* i number))
(mp:process-allow-scheduling)))
```

Note the use of mp:process-allow-scheduling to allow the process to be interrupted once during each iteration of the do loop.

Now we define the function that calls print-table within multiprocessing:

```
(defun process-print-table (name number total)
(mp:process-run-function name nil
  #'print-table number total *standard-output*))
```

The nil argument is used because no keywords are specified.

process-print-table can now be called from two separate Listener windows to print out different multiplication tables simultaneously, for example:

```
(process-print-table "t1" 5 50)
```

in one Listener and:

```
(process-print-table "t2" 6 50)
```

in another Listener.

11.3 Locks

Locks can be used to control access to shared data by several processes.

A lock has the following components: *name* (a string), *lock* (t or nil, that is, whether the lock is set or not), *owner* (a process, or nil) and *count* (an integer showing the number of times the lock has been set).

The two main symbols used in locking are the function make-lock, to create a lock, and the macro with-lock, to execute a body of code while holding the specified lock.

mp:make-lock

Function

mp:make-lock &key important-p &allow-other-keys

Creates a lock object. If important-p is t the lock is added to the list held in the global variable mp:*important-locks*. The function mp:freeimportant-locks frees all important locks associated with a given process (or all the important locks if called on nil). Other keywords should be names of the lock components.

mp:process-lock

mp:process-lock lock &optional whostate timeout

Blocks the current process until the lock is claimed or timeout elapses if it has been specified. Returns t if lock was claimed, nil otherwise.

mp:process-unlock

mp:process-unlock lock & optional errorp

Releases the lock. If *errorp* is non-nil it signals an error if the current process does not own the lock. The default value of *errorp* is t.

mp:with-lock

mp:with-lock ((lock &rest lock-args) &body body

Executes the body with lock held. Arguments to pass on to mp:process-lock are specified using *lock-args*.

The following accessors are available for locks: lock-owner, lock-count, lock-name and lock-lock.

11.4 Example

The following is an informal example of multi-processing with a single process (other then the idle process), namely a top-loop. Once it has started up, try (mp:ps).

Macro

Function

Function

```
(in-package "CL-USER")
;;; (guarantee-processes) will start up
;;; multiprocessing with a top-level loop
;;; in this example,
;;; use *base-process* to ensure that base
;;; process will only be pushed
;;; onto *initial-processes* once, no matter how
;;; many times guarantee-processes is called
(defvar *base-process*
 '("base-process" nil base-process-function))
;;; the base process consists of a top-level
;;; loops with restarts which allow control of
;;; return in the event of an error -- to see
;;; these in action, evaluate (guarantee-processes)
;;; and then an unbound variable.
;;; Note that starting and stopping multiprocessing is not
;;; relevant if Common LispWorks is already running. This example
;;; is included for illustration only.
(defun base-process-function ()
 (with-simple-restart
  (abort "Return from multiprocessing")
   (loop
    (with-simple-restart
     (abort "Return to top-level-loop")
     (system:%top-level))))
 (mp::stop-multiprocessing))
;;; simple startup of multiprocessing with one
;;; process (apart from the idle process)
(defun guarantee-processes ()
  (unless mp:*multiprocessing*
   (pushnew *base-process*
    mp:*initial-processes*)
   (mp:initialize-multiprocessing)))
```

Common Defsystem

12.1 Introduction

When an application becomes large, it is usually prudent to divide its source into separate files. This makes the individual parts of the program easier to find and speeds up editing and compiling. When you make a small change to one file, just recompiling that file may be all that is necessary to bring the whole program up to date.

The drawback of this approach is that it is difficult to keep track of many separate files of source code. If you want to load the whole program from scratch, you need to load several files, which is tedious to do manually, as well as prone to error. Similarly, if you wish to recompile the whole program, you must check every file in the program to see if the source file is out of date with respect to the object file, and if so re-compile it.

To make matters more complicated, files often have interdependencies; files containing macros must be loaded before files that use them are compiled. Similarly, compilation of one file may necessitate the compilation of another file even if its object file is not out of date. Furthermore, one application may consist of files of more than one source code language, for example Lisp files and C files. This means that different compilation and loading mechanisms are required.

The Common LispWorks system tools, and the system browser in particular, are designed to take care of these problems, allowing consistent development and maintenance of large programs spread over many files. A system is basically a collection of files that together constitute a program (or a part of a program), plus rules expressing any interdependencies which exist between these files.

You can define a system in your source code using the defsystem macro. Once defined, operations such as loading, compiling and printing can be performed on the system as a whole. The system tools ensure that these operations are carried out completely and consistently, without doing unnecessary work.

A system may itself have other systems as members, allowing a program to consist of a hierarchy of systems. Each system is treated independently of the others, and can be used to collect related pieces of code within the overall program. Operations on higher-level systems are invoked recursively on member systems.

12.2 Defining a system

A system is defined with a defsystem form in an ordinary Lisp source file. This form must be loaded into the Lisp image in order to define the system in the environment. Once loaded, operations can be carried out on the system by invoking Lisp functions, or, more conveniently, by using the system browser.

For example, the expression:

CL-USER 5 > (compile-system 'debug-app :force t)

would compile every file in a system called debug-app.

Note: When defining a hierarchy of systems, the leaf systems must be defined first — that is, a system must be declared before any systems that include it.

By convention, system definitions are placed in a file called defsys.lisp which usually resides in the same directory as the members of the system.

The full syntax of defsystem is given in the *LispWorks Reference Manual*. Below is a brief introduction.

12.2.1 DEFSYSTEM syntax

defsystem		Macro		
defsystem system-name options &key members rules				
system-name	A symbol used as the name of the system. If a str given, it is interned in the current package.	ing is		
options	Any of a number of options that can be specified.			
members	The members of the system. These may be files of mon Lisp source code, foreign source code, or oth tems.			
rules	A set of rules describing the requirements for and in which compilation and loading of the system r bers should take place.			

See the following sections for more information about these parameters.

12.2.2 DEFSYSTEM options

Options may be specified to defsystem which affect the behavior of the system as a whole. For example, :package specifies a default package into which files in the system are compiled and loaded if the file itself does not contain its own package declaration. The :default-pathname option tells the system tools where to find files which are not expressed as a full pathname.

12.2.3 DEFSYSTEM members

The :members keyword to defsystem is used to specify the members of a system. The argument given to :members is a list of strings. A system member is either a file or another system, identified by a name. If a full pathname is given then the function pathname-name is used to identify the name of the member. Thus, for example, the name of a member expressed as /u/neald/foo.lisp is foo.

The behavior of any member within a system can be constrained by supplying keyword arguments to the member itself. So, for example, specifying the

:source-only keyword ensures that only the source file for that member is ever loaded.

12.2.4 DEFSYSTEM rules

Rules may be defined in a system which modify the default behavior of that system, ensuring, for instance, that certain files are always loaded or compiled before others.

When you invoke an action such as compiling a system, the following happens by default:

- Each member of the system is considered in turn, in the order they are given in the system definition.
- If the member is itself a system then the action is performed on that system too, and so on recursively.
- If the member is a file and action-specific constraints are satisfied, the file action is inserted into a *plan*.

For example, in the case of compiling, a "compile this file" event is put into the plan if the source file is newer than the object file.

• After the plan has been assembled, it can be viewed or executed.

This behavior can be modified by describing dependencies between the members using *rules*. These are specified using the :rules keyword to defsystem.

A rule has three components:

The target(s). The action that is performed if the rule executes successfully. This is an action-member description like :compile "foo". The member can be an actual member of the system or :all (meaning the rule should apply to each member of the system).

The actions that the target(s) are :caused-by.

The actions that cause the rule to execute successfully.

This is a list of action-member descriptions. The member of each of these descriptions should be either a real system member, or :previous, which means all members listed before the member of the target in the system description.

If any of these descriptions are already in the current plan (as a result of other rules executing successfully, or as a result of default system behavior), they trigger successful execution of this rule.

The actions that the target(s) :requires.

The actions that need to be performed before the rule can execute successfully.

This is a list of action-member descriptions that should be planned for before the action on the target(s). Again, each member should either be a real member of the system, or :previous.

The use of the keyword :previous means, for example, that you can specify that in order to compile a file in the system, all the members that come before it must be loaded.

When the action and member of a target are matched during the traversal of the list of members, the target is inserted into the plan if either of the following are true:

- any of the action-member descriptions in the :caused-by clause is already in the plan, or
- any implicit conditions (such as the source file being newer than the object file) are satisfied.

If the target is put into the plan then other targets are inserted beforehand if the action-member description of any **:requires** clause is not already in the plan.

12.2.5 Examples

Consider an example system, demo, defined as follows:

```
(defsystem demo (:package "USER")
  :members ("parent"
        "child1"
        "child2")
  :rules ((:in-order-to :compile ("child1" "child2")
        (:caused-by (:compile "parent"))
        (:requires (:load "parent")))))
```

This system compiles and loads members into the USER package if the members themselves do not specify packages. The system contains three members — parent, child1, and child2 — which may themselves be either files or other systems. There is only one explicit rule in the example. If parent needs to be compiled (for instance, if it has been changed), then this causes child1 and child2 to be compiled as well, irrespective of whether they have themselves changed. In order for them to be compiled, parent must first be loaded.

Implicitly, it is always the case that if any member changes, it needs to be compiled when you compile the system. The explicit rule above means that if the changed member happens to be parent, then *every* member gets compiled. If the changed member is not parent, then parent must at least be loaded before compiling takes place.

The next example shows a system consisting of three files:

```
(defsystem my-system
 (:default-pathname "~/junk/")
  :members ("a" "b" "c")
  :rules ((:in-order-to :compile ("c")
                    (:requires (:load "a"))
                    (:caused-by (:compile "b")))))
```

What plan is produced when all three files have already been compiled, but the file **b.lig** has since been changed?

First, file a.lisp is considered. This file has already been compiled, so no instructions are added to the plan.

Second, file **b.lisp** is considered. Since this file has changed, the instruction *compile b* is added to the plan.

Finally file c.lisp is considered. Although this has already been compiled, the clause

(:caused-by (:compile "b"))

causes the instruction *compile c* to be added to the plan. The compilation of c.lisp also requires that a.lisp is loaded, so the instruction *load a* is added to the plan first. This gives us the following plan:

- 1. Compile b.lisp.
- 2. Load a.lisp.
- 3. Compile c.lisp.

Common Defsystem

The Parser Generator

13.1 Introduction

The parser generator generates an LALR parser from a specification of a grammar. The parser generator has a simple facility for the static resolution of ambiguity in the grammar and supports an automatic run-time error correction mechanism as well as user-defined error correction. Semantic actions can be included in the rules for the grammar by specifying Lisp forms to be evaluated when reductions are performed.

For further details on LALR parsing, see *Compilers, Principles Techniques and Tools*, by Aho, Sethi and Ullman, publishers Addison Wesley, 1986.

13.2 Grammar rules

The parser generator is accessed by the macro defparser, described below:

defparser

Macro

```
defparser name {rules}*
```

name The name to be used for the parsing function. The remainder of the macro form specifies the reduction rules and semantic actions for the grammar.

rules The rules specified in a defparser form are of two types, *normal rules* and *error rules*, described below.

Each normal rule corresponds to one production of the grammar to be parsed:

```
((non-terminal {grammar-symbol}*) {form}*)
```

The *non-terminal* is the left-hand side of the grammar production and the list of grammar symbols defines the right-hand side of the production. (The right-hand side may be empty.) The list of forms specifies the semantic action to be taken when the reduction is made by the parser. These forms may contain references to the variables $\$1 \dots \n , where n is the length of the right hand side of the production. When the reduction is done, these variables are bound to the semantic values corresponding to the grammar symbols of the rule.

13.2.1 Example

If a grammar contains the production:

```
expression -> expression operator expression
```

with a semantic representation of a list of the individual semantic values, the Lisp grammar would contain the rule:

```
((expression expression operator expression) (list $1 $2 $3))
```

Error productions of the form:

((nt :error) (some error behavior))

are explained in the section below.

The first rule of the grammar should be of the form:

((*nt* nt1) \$1)

where the non-terminal *nt* has no other productions and **nt1** serves as the main "top-level" non-terminal.

13.2.2 Resolving ambiguities

If the grammar is ambiguous, there is conflict between rules of the grammar: either between reducing with two different rules or between reducing by a rule and shifting an input symbol. Such a conflict is resolved at parser generation time by selecting the highest priority action, where the priority of a reduce action is determined by the closeness of the rule to the beginning of the grammar. A priority is assigned to a shift by associating it with the rule that results in the shift being performed.

For example, if the grammar contains the two rules:

- Rule a: statement -> :if expression :then statement :else statement
- Rule b: statement -> :if expression :then statement

this results in a conflict in the parser between a shift of :else, for rule a, and a reduce by rule b. This conflict may be resolved by listing rule a earlier in the grammar than rule b. This ensures that the shift is always done.

Note that ambiguities cannot always be resolved successfully in this way. In this example, if the ambiguity is resolved the other way around, by listing rule b first, this results in the if ... then ... part of an if ... then ... else ... statement being reduced, and a syntax error is produced for the else part.

During parser generation, any conflicts between rules are reported, together with information about how the conflict was resolved.

13.3 Functions defined by defparser

The form (defparser name grammar) defines two main functions. The function <*name*> is defined as the parsing function, and the function <*name*>grammar is a parameterless function that returns the grammar, as presented to the parser generator. For example:

```
(defparser my-parser .. grammar .. )
```

defines the two functions:

```
(defun my-parser (lexer &optional
                                 (symbol-to-string #'identity)) ...)
```

and

```
(defun my-parser-grammar () ..)
```

The lexer parameter to the parser function specifies the lexical analyzer function to be used. The optional symbol-to-string function may be used to define a mapping from grammar symbols to strings for printing purposes. It defaults to the identity function.

defparser also defines functions corresponding to the individual actions of the parser.

Normal actions are named:

<parser-name>-action<index>

Error actions are named:

<parser-name>-error-action<index>

parser-name here is the name as given to defparser. *index* is the number of the rule or error rule in the grammar.

All function names are interned in the current package when defparser is called.

13.4 Error handling

The parser supports automatic error correction of its input. The strategy used involves attempting to either push a new token onto the input, replacing an erroneous symbol, or discarding an erroneous symbol. Such action is only taken if it is guaranteed that the parser can continue parsing and read at least one more symbol from its input.

If the correction strategy fails, then error recovery is invoked.

The parser allows the inclusion of grammar productions of the form:

non-terminal -> :error

This means that the parser accepts an erroneous string of tokens as constituting an occurrence of the non-terminal. Such productions may be used to skip over portions of input when attempting to recover from an error. The action associated with such an error is specified by a form in the same way as for ordinary actions. The action may perform manipulation of the parser state and input.

13.5 Interface to lexical analyzer

The lexical analyzer function that is passed to the parser is expected to be a function of zero arguments that returns two values each time it is called. The first value is the next token on the input and the second value is the semantic value corresponding to that token. If there is no more input, then the lexical analyzer may return either the token :eoi or nil.

For example:

```
(defparser my-parser
   ...)
(defun my-lexer (stream)
   .. read next token from stream ..
  (values token value))
(defun my-symbol-to-string (symbol)
   .. returns a string ..)
(defun my-parse-stream (stream)
  (let ((lexer #'(lambda () (my-lexer stream))))
    (my-parser lexer #'my-symbol-to-string)))
```

Note that during error correction, the parser may push extra tokens onto the input, in which case they are given the semantic value nil. The semantic actions should therefore be capable of dealing with this situation. Manipulation of the input (e.g. pushing extra tokens) is done within the parser generator and the lexical analyzer need not concern itself with this.

13.6 Example

The following example shows a simple grammar for a very small subset of English.

```
(defpackage "ENGLISH-PARSER")
(in-package "ENGLISH-PARSER")
(use-package '(parsergen))
;;; Define the parser itself.
(defparser english-parser
  ((bs s) $1)
  ((s np vp)
  `(,$1 ,$2))
  ((bnp :adj bnp)
  `(,$1 ,$2))
  ((bnp bnp relp)
  `(,$1 ,$2))
  ((bnp :noun) $1)
  ((relp :rel vp)
  `(,$1 ,$2))
  ((vp :verb np locp)
  `(,$1 ,$2 ,$3))
  ((vp :verb locp)
  `(,$1 ,$2))
  ((vp :verb np)
  `(,$1 ,$2))
  ((vp :verb)
  $1)
  ((np :art bnp locp)
  `(,$1 ,$2 ,$3))
  ((np :art bnp)
  `(,$1 ,$2))
  ((np bnp) $1)
  ((locp :loc np)
   `(,$1 ,$2)))
;;; The lexer function.
;;; The basic lexing function
(defvar *input*)
(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol)
     nil)))
;;; Getting syntactic categories.
(defparameter *words*
 '((the :art)(a :art)(some :art)(ate :verb)(hit :verb)
```

```
(cat :noun)(rat :noun)(mat :noun)(which :rel)(that :rel)
(who :rel)(man :noun)(big :adj)(small :adj)(brown :adj)
(dog :noun)(on :loc)(with :loc)(behind :loc)(door :noun)
(sat :verb)(floor :noun)))
(defun get-lex-class (word)
  (values
    (or (cdr (assoc word *words*))
        :unknown)
    word))
;;; The main function -- note bindings of globals (these ;;; are
exported from the parsergen package).
(defun parse-english (input)
  (let ((*input* input))
        (english-parser #'lex-english)))
```

The following example session shows the parsing of some sentences.

The Parser Generator

Dynamic Data Exchange

14.1 Introduction

Dynamic data exchange (DDE) involves passing data and instructions between applications running under the Windows operating system. Typically the data is passed in the form of a string, which is interpreted when it is received. One application acts as a *server* and the other as a *client*.

14.1.1 Types of transaction

The server is normally a passive object, which waits for a client object to tell it what to do. The client can communicate with the server in four ways:

- The client can issue a *request transaction* to the server. This means the client is asking for some information about the server application.
- The client can issue a *poke transaction*. This means the client is passing data to be stored by the server application.
- The client can issue an *execute transaction*. This means the client is asking the server to get the server application to run a command.
- The client can ask the service to set up an *advise loop*, or to close an existing advise loop. An advise loop causes the server to communicate with the client whenever a specified change occurs in the server application.

14.1.2 Conversations, servers, topics, and items

For a transaction to take place between a client and a server, a conversation must be established. A conversation is established when a client makes a request by broadcasting a service name and topic name, and a server responds. Transactions can then take place across the conversation. When no more transactions are to be made, the conversation is terminated.

The following list identifies the elements involved with client/server activity:

conversation	A conversation is established when a server responds to a client.
service name	A service name is a string broadcast by a client hoping to establish a conversation with a server that recognizes the service name. The service name is usually clearly related to the server application name.
topic name	The topic name identifies what the conversation between client and server is to be about. For example, it could be the name of a file that is open in the server application. Each topic is attached to one particular server. A server can have many topics.
item name	The item usually identifies an element of the file identi- fied by the topic which should be read (in the case of a request) or written to (in the case of a poke). For exam- ple, it might refer to a cell in a spreadsheet document.

14.1.3 Advise loops

An advise loop instructs the server to inform the client when data in the server's application changes. Advise loops are set up across a conversation, and closing the conversation closes the advise loop.

An advise loop is identified by an item and a key. The key is included to allow any number of uniquely identifiable advise loops to be set up on the same server/topic/item combination.

A successfully established advise loop is also known as a link. When a change occurs to item, the link informs the client by causing it to execute a function.

There are two types of link: the warm link which only informs the client that a change to item has occurred, and the hot link which also sends the new data across.

14.1.4 Execute transactions

When a client issues an execute transaction to a server, the command to be executed is transferred as a string. This involves the marshalling of the command and its arguments into a suitable string format. The standard format of such a string is:

```
[command(arg1,arg2,...)]
```

14.2 Client interface

14.2.1 Opening and closing conversations

A LispWorks client can open a conversation by using dde-connect, which takes a service designator and a topic designator as its arguments. If successful, a conversation object is returned which can be used to refer to the conversation. Conversations are closed by the LispWorks client at the end of a transaction by using dde-disconnect.

dde-connect	Function
dde-connect service topic akey class errorp	
The function dde-connect attempts to create a conversation with server specified by <i>server</i> , on the topic given by <i>topic</i> .	ı a DDE
dde-disconnect	Function
dde-disconnect conversation	
The function dde-disconnect disconnects the conversation object field by <i>conversation</i> . The conversation may no longer be used.	ct speci-
Another method for managing conversations uses with-dde-conversation bind a conversation with a server across a body of code. If no convers	

available for with-dde-conversation, then one is automatically opened, *body* is executed, and then the conversation is closed.

with-dde-conversation

Macro

with-dde-conversation (conv service topic &key errorp new-conversation-p) &body body

The macro with-dde-conversation dynamically binds a conversation with a server across the scope of a body of code specified by *body*. The argument *conv* is bound to a conversation with the server specified by *service*, and the topic specified by *topic*.

14.2.2 Automatically managed conversations

There is an alternative to manually establishing a conversation and then disconnecting it once all transactions between server and client are concluded: the automatically managed conversation. Client functions that end with a * conduct automatically managed conversations.

A function handling an automatically managed conversation takes a service designator and topic designator as two of its arguments, and either automatically establishes a conversation with a server responding to the service designator/topic designator pair, or uses an existing equivalent conversation. For the purpose of brevity, functions conducting automatically managed conversations will not be explicitly documented in this chapter. See the *LispWorks Reference Manual* for full details.

14.2.3 Advise loops

A LispWorks client can set up an advise loop across a conversation using dde-advise-start, which takes a *conversation* (or a *service* designator/*topic* designator pair in the case of an automatically managed conversation using dde-advise-start*), an *item*, and a *key* as its main arguments. The *key* argument defaults to the conversation name, and can be used to distinguish between multiple advise loops established on the same service/topic/item group.

Function

dde-advise-start conversation item &key key function format datap type errorp

The dde-advise-start function sets up an advise loop for the data item specified by *item* on the specified *conversation*.

dde-client-advise-data

dde-advise-start

dde-client-advise-data key item data &key &allow-other-keys

The generic function dde-client-advise-data is the default function called when an advise loop informs a client that the data monitored by the loop has changed. By default it does nothing, but it may be specialized on the object used as the key in dde-advise-start or dde-advisestart*, or on a client conversation class if the default key is used.

define-dde-client

define-dde-client name &key service class

The macro define-dde-client defines a mapping from the symbol name to the DDE service name with which to establish a conversation, and the conversation class to use for this conversation. The argument *service* is a string which names the DDE service. It defaults to the print-name of *name*. The argument *class* is a subclass of dde-client-conversation which is used for all conversations with this service. It defaults to ddeclient-conversation. Specifying a subclass allows various aspects of the behavior of the conversation to be specialized.

The following is an example of how to set up an advise loop. The first step defines a client conversation class, called my-conv.

(defclass my-conv (dde-client-conversation)
 ())

The function define-dde-client can now be used to define a specific instance of the my-conv class for referring to a server application that responds to the service name "FOO".

(win32:define-dde-client :foo :service "FOO" :class my-conv)

Macro

Generic Function

The next step defines a method on dde-client-advise-data which returns a string stating that the item has changed.

```
(defmethod dde-client-advise-data ((self my-conv) item data &key
&allow-other-keys)
  (format t "~&Item ~s changed to ~s~%" item data))
```

Finally, the next command starts the advise loop on the server foo, with the topic name "file1", to monitor the item "slot1".

(win32:dde-advise-start* :foo "file1" "slot1")

When the value of the item specified by "slot1" changes, the server calls ddeclient-advise-data which returns a string, as described above.

The function argument of dde-advise-start and dde-advise-start* specifies the function called by the advise loop when it notices a change to the item it is monitoring. The function is dde-client-advise-start by default. A different function can be provided, and should have a lambda list similar to the following:

```
key item data &key conversation &allow-other-keys
```

The arguments *key* and *item* identify the advise loop, or link. The argument *data* contains the new data for hot links; for warm links it is nil.

Advise loops are closed using dde-advise-stop or dde-advise-stop*.

dde-advise-stop

Function

dde-advise-stop conversation item &key key format errorp disconnectp no-advise-ok

The function dde-advise-stop removes a particular link from *conversation* specified by *item*, *format* and *key*. If *key* is the last key for the *item/ format* pair, the advise loop for the pair is terminated.

14.2.4 Request and poke transactions

LispWorks clients can issue request and poke transactions across a conversation using dde-request and dde-poke, which take a *conversation* (or a *service* designator/*topic* designator pair in the case of an automatically managed conversation), and an *item* as their main arguments. In the case of a poke transaction, data to be poked into *item* must also be provided.

In the case of a successful request transaction with dde-request or dderequest*, the data contained in *item* is returned to the LispWorks client by the server.

In the case of a successful poke transaction with dde-poke or dde-poke*, the data provided is poked into *item* by the server.

The accessor dde-item (or dde-item* for automatically managed conversations) can perform request and poke transactions. See the *LispWorks Reference Manual* for more details.

dde-item

dde-item conversation item &key format type errorp

The accessor dde-item performs a request transaction when read. It performs a poke transaction when set.

dde-poke

dde-poke conversation item data &key format type errorp => result

The function dde-poke issues a poke transaction on *conversation* to set the value of the item specified by *item* to the value specified by *data*. The argument *item* should be a string, or a symbol. If it is a symbol its print name is used.

dde-request

dde-request conversation item &key format type errorp

The function dde-request issues a request transaction on *conversation* for the specified *item*. The argument *item* should be a string, or a symbol. If it is a symbol its print name is used.

Function

Accessor

Function

14.2.5 Execute transactions

A client can issue an execute transaction across a conversation, or in the case of an automatically established conversation, to a recognized server. There is no need to specify a topic, as an execute transaction instructs the server application to execute a command.

The command and its arguments are issued to the server in the form of a string in a standard format (see "Execute transactions" on page 111). Lisp-Works provides two ways of issuing an execute transaction, namely dde-execute-string and dde-execute-command (and the corresponding * functions that automatically manage conversations).

dde-execute-string

Function

dde-execute-string conversation command &key errorp

The function dde-execute-string takes the command to issue in the form of an appropriately formatted string. The following example shows how dde-execute-string* can issue a command to a server designated by :excel on the topic :system, in order to open a file called foo.xls:

(win32:dde-execute-string* :excel :system "[open(\"foo.xls\")]")

dde-execute-command

Function

dde-execute-command conversation command arg-list &key errorp

The function dde-execute-command takes the command to issue, and its arguments, and marshals these into an appropriate string for you. The following example shows how dde-execute-command* can issue the same command as in the previous example:

(win32:dde-execute-command* :excel :system `open `("foo.xls"))

14.3 Server interface

14.3.1 Starting a DDE server

To provide a LispWorks application with a DDE server, the following three steps should be followed: define a specialized server class using define-ddeserver, provide the server class with the functionality it requires by specializing methods on it and/or using define-dde-server-function, and finally, start an instance of the server using start-dde-server

define-dde-server

define-dde-server class-name service-name

The macro define-dde-server defines a class for a Lisp DDE server. The class inherits from dde-server.

define-dde-server-function

define-dde-server-function name-and-options transaction (binding*) form*

The macro define-dde-server-function is used to define a server function, called *name*, which is called when a specific transaction occurs.

The defined function may either be attached to a server object (possibly only for a particular topic class) or to a dispatching topic object.

start-dde-server

start-dde-server name

The function start-dde-server creates an instance of a server of the class specified by *name* which then starts accepting transactions. If successful the function returns the server. otherwise nil is returned.

The next command line shows how to use define-dde-server to define a server class called foo-server that has the service name "FOO".

(win32:define-dde-server foo-server "FOO")

Macro

Function

Macro

It is usual to provide the new server class with some functionality. The next command illustrates how to define a server function called test, which takes a string as an argument, and prints this to the standard output.

```
(win32:define-dde-server-function test ((x string))
 (format t "~&~s~%" x)
 t)
```

Finally, a foo-server can be started using start-dde-server

(win32:start-dde-server `foo-server)

This function returns the server object, which responds to requests for conversations with the service name "FOO", and accepts execute transactions for the function test.

14.3.2 Handling poke and request transactions

Poke and request transactions issued to a server object are handled by defining a method on each of the generic functions dde-server-poke and ddeserver-request.

dde-server-poke

Generic Function

dde-server-poke server topic item data &key format &allow-other-keys

The generic function dde-server-poke is called in response to a poke transaction. A method specializing on the classes of *server* and *topic* should poke the data given by *data* into the item specified by *item*.

dde-server-request

Generic Function

dde-server-request server topic item &key format &allow-other-keys

The generic function dde-server-request is called in response to a request transaction. A method specializing on the classes of *server* and *topic* should return the data in *item*.

14.3.3 Topics

DDE servers respond to connection requests containing a service name and a topic name. The service name of a server is the same for any conversation whereas the topic name may vary from conversation to conversation, and identifies the context of the conversation. Typically, valid topics correspond to open documents within the application, so the set of valid topics varies from time to time. In addition, all servers implement a topic called "system", which contains a standard set of items that can be read.

The LispWorks DDE interface supports three types of topics:

1. General topics

A general topic is an instance of a user-defined topic class. The actual set of topics available may vary from time to time as the application is running.

2. Dispatching topics

A dispatching topic has a fixed name, and is available at all times that the server is running. It supports a fixed set of items, and each of these items has Lisp code associated with it to implement these items.

3. The system topic.

The system topic is provided automatically by the LispWorks DDE interface. However, a mechanism is provided to extend the functionality of the system topic by handling additional items.

14.3.3.1 General topics

To use general topics, the LispWorks application must define one or more subclasses of dde-topic. If an application supports only a single type of document, it will typically require only one topic class. If several different types of document are supported, it may be convenient to define a different topic class for each type of document.

If the application uses general topics, it should define a method on the ddeserver-topics generic function, specializing on the application's server class.

dde-server-topics

Generic Function

dde-server-topics server

The generic function dde-server-topics returns a list of the available general topics on a given server. A suitable method specializing on the server class should be defined. Dispatching topics should not be returned, as they are handled automatically by LispWorks. If you do not provide a dde-server-topics method, the default method returns :unknown, which prevents the DDE server from responding to the topics request.

14.3.4 Dispatching topics

A dispatching topic is a topic which has a fixed name and always exists. Dispatching topics provide dispatching capabilities, whereby appropriate application-supplied code is executed for each supported transaction. Dispatch topics are defined using define-dde-dispatch-topic.

define-dde-dispatch-topic

Macro

define-dde-dispatch-topic name &key server topic-name

The macro define-dde-dispatch-topic defines a dispatching topic. Note that the server implementation also provides some dispatching capabilities.

14.3.5 The system topic

The system topic is implemented as a predefined dispatching topic called :system. It is automatically available to all defined DDE servers. Its class is dde-system-topic, which is a subclass of dde-dispatching-topic.

The following items are implemented by the system topic:

SZDDESYS_ITEM_TOPICS

Constant

The constant szddesys_ITEM_TOPICS has the value "Topics". Referring to this item in the system topic calls dde-server-topics to obtain a list of topics implemented by the server. The server should define a method on this generic function to return a list of strings naming the topics supported by the server. If this item is not to be implemented, do not define a method on the function, or define a method that returns :unknown.

SZDDESYS_ITEM_SYSITEMS

The constant szDDESYS_ITEM_SYSITEMS has the value "SySItems". Referring to this item in the system topic calls dde-topic-items to obtain a list of items implemented by the system topic. If a server implements additional system topic items it should define a method on the generic function specialized on its server class and dde-system-topic returning the complete list of supported topics. The server can return :unknown if this item is not to be implemented.

SZDDESYS_ITEM_FORMATS

The constant szddesys_ITEM_FORMATS has the value "Formats", and returns unicodetext and text. Currently only text formats are supported.

The system topic is a single object which is used by all DDE servers running in the Lisp image. You should therefore not under normal circumstances modify it with define-dde-server-function by specifying a value of :system for the *topic* argument, as this would make the changes to the system topic visible to all users of DDE within the Lisp image.

Instead, specify :server my-server :topic :system, where my-server is the name of your DDE server. This makes the additional items available only on the system topic of the specified server.

Constant

Constant

Dynamic Data Exchange

15

Common SQL

15.1 Introduction

This chapter describes Common SQL — the LispWorks interface to SQL. It should be used in conjunction with the relevant chapter of the *LispWorks Reference Manual*, which contains full reference entries for all the symbols in the SQL package. This chapter is applicable to the Enterprise version of LispWorks only. This chapter covers the following areas:

- Initialization and Connection
- The Functional SQL Interface
- The Object-Oriented (CLOS) SQL Interface
- The Symbolic SQL Syntax
- SQL I/O Recording
- SQL Interface Errors

The LispWorks SQL interface uses the following database terminology:

Data Definition Language (DDL)

The language used to specify and interrogate the structure of the database schema.

Data Manipulation Language (DML)

	The language used for retrieving and modifying data. Also known as <i>query language</i> .
table	A set of records. Also known as <i>relation</i> .
attribute	A field of information in the table. Also known as <i>column</i> .
record	A complete set of attribute values in the table. Also known as <i>tuple</i> , or <i>row</i> .
view	A display of a table configured to your own needs. Also known as <i>virtual table</i> .

15.1.1 Overview

Common SQL is designed to provide both embedded and transparent access to relational databases from the LispWorks environment. That is, SQL/relational data can be directly manipulated from within Lisp, and also used as necessary when instantiating or accessing particular Lisp objects.

The SQL interface allows the following:

- Direct use of standard SQL statements as strings
- Mixed symbolic SQL and Common Lisp expressions
- Implicit SQL invocation when instantiating or accessing CLOS objects

The SQL interface provides these features through two complementary layers:

- A functional SQL interface
- An *object-oriented* SQL interface

The functional interface provides users with Lisp functions which map onto standard SQL DML and DDL commands. Special iteration constructs which utilize these functions are also provided. The object-oriented interface allows users to manipulate database views as CLOS classes via def-view-class. The two interfaces may be flexibly combined in accordance with system requirements and user preference. For example, a *select* query can be used to initialize slots in a CLOS instance; conversely, accessing a CLOS slot may trigger an implicit functional query.

15.2 Initialization

The initialization of Common SQL usually involves several stages. Firstly the SQL interface itself is initialized. Then, optionally, the SQL package can be loaded. Next, the database types to be used are initialized. Finally, Common SQL is used to connect to a database. These stages are explained in more detail in this section.

The rest of the definitions in this chapter are exported from the sql package. Application packages requiring convenient access to these facilities should therefore use the sql package.

15.2.1 SQL interface

The SQL interface itself is initialized by issuing the command (require "odbc"). Only ODBC database types are currently supported by LispWorks for the Windows operating system.

15.2.2 Database classes

A connection to a database is represented by an instance of the CLOS class sql:database. This instance holds information about the connected database. The special variable sql:*default-database* holds the current connection. The database class is subclassed on both vendor and version to provide the right kind of specialized behavior across database facilities: for example, the transaction model or the "brand" of SQL.

15.2.3 Initialization functions and variables

The generic SQL interface code, including the SQL package itself, and the appropriate libraries, are load-on-demand. The initialization of the ODBC database type is achieved by calling sql:initialize-database-type on the database type :odbc, after you have initialized SQL by requiring ODBC.

The following functions and variables are relevant to initialization:

default-database-type

Specifies the default type of database. Currently only ODBC is supported, and therefore this should be set to :odbc.

initialized-database-types

Contains a list of database types which have been initialized by calls to initialize-database-type.

initialize-database-type

Function

Variable

Variable

sql:initialize-database-type &key database-type

Initializes a database type by loading code and appropriate database libraries according to the value of *database-type*. Adds *database-type* to the list of initialized types. The sql package itself is loaded during the first call to this function. A sample code sequence for initializing SQL to work with an ODBC database, using the above functions and variables, is as follows:

```
(require "odbc")
(in-package "sql")
(setf *default-database-type* :odbc)
(initialize-database-type)
```

15.2.4 Database connection and disconnection

Once the database type has been initialized a connection can be established by calling connect. A call to connect sets *default-database* to the database instance which represents the connection. All the other database functions described take a :database argument that can be either a database or a database name, and which defaults to *default-database*.

default-database

Variable

Specifies the default database to be used for database operations.

connected-databases

Returns a list of database connection instances.

connect

Function

Function

connect connection-spec &key if-exists database-type

Opens a connection to a database of *database-type*. The argument *connection-spec* depends on the type of the connected database. For databases of type **:odbc**, the connection specification is a string of the format

"datasource-name/username/password"

where *datasource-name* is the name of an ODBC datasource, and *username* and *password* are a valid username and password. If the datasource does not require a username and password they can be omitted.

The connect function sets *default-database* to an instance of the database opened and returns that instance. The argument *if-exists* modifies the behavior of connect as follows:

:new	Makes a new connection even if connections to the same database already exist.
:warn-new	Makes a new connection but warns about existing con- nections.
:error	Makes a new connection but signals an error for exist- ing connections.
:warn-old	Selects the old connection if one exists (and warns) or makes a new one.
:old	Selects the old connection if one exists or makes a new one.

connect-if-exists

Variable

Default value for the **:if-exists** variable of the connect function. See the connect description for the possible values. Initial value is **:error**.

disconnect

disconnect &key database Closes the connection to database. Resets *default-database* if that database was disconnected and only one other connection exists. database-name database-name database Returns the database connection string. find-database find-database database & optional errorp

Returns a database given its name or itself. If database is not found among the connected databases and errorp is non-nil then find-database signals an error, otherwise it returns nil. By default, errorp is t.

status

Function

Function

Function

Function

status & optional full

Returns status information for the connected databases and initialized database types. By default, full is nil. If t, more detailed information is returned.

15.2.4.1 Connection example

The following example assumes that the sql package has been loaded, and that the :odbc database type has been initialized. It connects to two databases, scott and personnel, and then prints out the connected databases.

```
(setf *default-database-type* :odbc)
(connect "scott")
(connect "personnel" :database-type :odbc)
(print *connected-databases*)
```

15.3 Functional interface

The functional interface provides a full set of Data Manipulation and Data Definition functions. The interface provides an SQL-compatible means of querying and updating the database from Lisp. In particular, the values returned from the database are Lisp values — thus smoothly integrating user applications with database transactions. An embedded syntax is provided for dynamically constructing sophisticated queries through select. Iteration is also provided via a mapping function and an extension to the loop macro. If necessary, the basic functions query and execute-command can be called with SQL statements expressed as strings. It is also possible to update or query the data dictionary.

15.3.1 Functional Data Manipulation Language (FDML)

The functions available for Data Manipulation and Data Definition are described below.

15.3.1.1 Querying

select

Function

select &rest selections &key all set-operation distinct from where flatp
 group-by having order-by database

Selects data from *database* given the constraints specified. Returns a list of lists of record values as specified by *args*. By default, the records are each represented as lists of attribute values. The *selections* argument may be either db-identifiers or literal strings.

Database identifiers used in select are conveniently specified using the symbolic SQL [] syntax. This syntax is enabled by calling enable-sql-readersyntax, and is only available in packages which inherit from the SQL package.

You should therefore create a new package which includes the sql package, before using the bracket syntax.

For a description of the symbolic SQL syntax see Section 15.5 on page 143. For example, the following is a potential query and result:

```
(select [person_id] [person surname] :from [person])
-> ((111 "Brown") (112 "Jones") (113 "Smith"))
```

In this example, [person_id], [person surname] and [person] are databaseidentifiers and evaluate to literal SQL. The result is a list of lists of attribute values. Conversely, consider

```
(select [surname] :from [person] :flatp t)
-> ("Brown" "Jones" "Smith")
```

In this case the result is a simple list of surname values because of the use of the *flatp* keyword. The *flatp* keyword only works when there is one column of data to return.

In this final example the :where keyword is used to specify a condition for returning selected values from the database.

```
(select [surname] :from [person] :where [= [person_id] 112])
-> (("Jones"))
```

print-query

Function

print-query query-exp &key titles formats sizes stream database

Prints a tabulated version of the records resulting from query-exp.

titles	A list of strings for using as column headings — nil means no column headings are used. It is nil by default.
formats	A list of format strings used to print each attribute — t means use ~a, or ~va if sizes are provided or computed. It is t by default.
sizes	A list of field sizes for printing the attributes — t means compute minimum sizes. It is t by default.
stream	An output stream — t means use *standard-output*. It is t by default.

The following call prints two even columns of names and salaries:

```
(print-query [select [surname] [income] :from [person]]
                :titles '("NAME" "SALARY"))
NAME SALARY
Brown 22000
Jones 45000
Smith 35000
```

15.3.1.2 Modification

Modifications to the database can be done using the following functions; insert-records, delete-records and update-records. The functions commit, rollback and with-transaction are used to control transactions. Although commit or rollback may be used in isolation it is advisable to do any updates inside a with-transaction form instead. This provides consistency across different database transaction models. For example, some database systems do not provide an explicit "start-transaction" command while others do. The with-transaction form allows user code to ignore databasespecific transaction models.

insert-records

Function

insert-records &key into attributes values av-pairs query database

Inserts values for attributes (or av-pairs) into the table into.

values A list of values or a query expression.

av-pairs A list of two-element lists of attributes and values.

For example:

(insert-records :into [person]
 :values
 '(114 "Joe" "Bloggs" 10000 3000 nil "plumber"))

is equivalent to the following SQL:

```
INSERT INTO PERSON
VALUES (114,'Joe','Bloggs',10000,3000,NULL,'plumber')
```

If *attributes* is supplied then *values* must be a corresponding list of values for each of the listed attribute names. For example, both:

are equivalent to the following SQL:

```
INSERT INTO PERSON
 (PERSON_ID,INCOME,SURNAME,OCCUPATION)
 VALUES (115,11000,'Johnson','plumber')
```

If *query* is provided, then neither *values* nor *attributes* should be. In this case the attribute names in the query expression must also exist in the insertion table. For example:

```
(insert-records :into [person]
  :query [select [id] [firstname] [surname]
                         :from [manager]]
                    :attributes '(person_id firstname surname))
```

delete-records

Function

delete-records &key from where database

Deletes rows from table *from* where the *where* condition is true.

update-records

Function

update-records table &key attributes values av-pairs where database

Changes the values of fields in *table* with columns specified by *attributes* and *values* (or by *av-pairs*) where the *where* condition is true.

15.3.1.3 Transaction handling

A transaction in SQL is defined as *starting from* the connect, or from a commit, rollback or data-dictionary update and *lasting until* a commit, rollback, data-

Macro

dictionary update or a disconnect command. Applications should perform all database update operations in a with-transaction form in order to safely commit their changes.

The following example shows a series of updates to an employee table within a transaction. This example would commit the changes to the database on exit from with-transaction. This example inserts a new record into the emp table, then changes those employees whose department number is 40 to 50 and finally removes those employees whose salary is more than 300,000.

```
(connect "personnel")
(with-transaction
  (insert-records :into [emp]
                    :attributes '(empno ename job deptno)
                    :values '(7100 "ANDERSON" "SALESMAN" 30))
(update-records [emp]
                    :attributes [deptno]
                    :values 50
                    :where [= [deptno] 40])
(delete-records :from [emp]
                    :where [> [sal] 300000]))
```

with-transaction

with-transaction &key database &body body

Performs *body* within a transaction for *database*. The transaction is committed if the body finished successfully (without aborting or throwing), otherwise the database is rolled back.

rollback Function rollback &key database Rolls back changes made in database since the last commit. commit commit &key database

Commits changes made to database.

134

Sometimes it is necessary to execute vendor-specific SQL statements and queries. For these occasions we provide the two functions below. They can also be used when the exact SQL string is known in advance and thus the square bracket is not required.

query

Common SQL

query query-expression &key database

Basic SQL query function which queries *database* with *query-expression* and returns a list of values as per select. This and execute-command should be used to execute non-standard vendor specific SQL code.

execute-command

execute-command sql-expression &key database

Basic function which executes the *sql-expression*. The *sql-expression* may be any SQL statement other than a query.

15.3.1.4 Iteration

Common SQL has three iteration constructs: a do-loop, a mapping function, and an extension to the Common Lisp loop macro.

do-query

do-query &rest args query &key database &rest body

Repeatedly executes *body* within a binding of *args* to the attributes of each record resulting from *query*.

map-query

map-query result-type function query-expression &key database

Returns the result of mapping *function* across the results of *query-expression*. The *result-type* argument specifies the type of the result sequence as per the Common Lisp map function.

Functions

Function

Function

Macro

loop

Macro

The Common Lisp loop macro has been extended with a clause for iterating over query results. The syntax of the new clause is:

```
{for|as} var [type-spec] being
        {the|each}{tuples|tuple}
        {in|of} query-expression
```

The more general word tuple is used so that it can also be applied to the object-oriented case. In the functional case, tuple is synonymous with record.

Each iteration of the loop assigns the next record of the table to the variable *var*. The record is represented in Lisp as a list. Destructuring can be used in *var* to bind variables to specific attributes of the records resulting from *query-expression*. In conjunction with the panoply of existing clauses available from the loop macro, the new iteration clause provides an integrated report generation facility.

Suppose the name of everyone in an employee table is required. This simple query is shown below using the different iteration method. The function map-query requires *flatp* to be specified; otherwise each name would be wrapped in a list.

```
(do-query ((name)[select [ename] :from [emp]])
        (print name))
(map-query
      nil
      #'(lambda (name) (print name))
      [select [ename] :from [emp] :flatp t])
(loop for (name)
           being each tuple in
              [select [ename] :from [emp]]
                do
(print name))
```

The following extended loop example binds, on each record returned as a result of the query, name and salary, accumulates the salary, and for salaries greater than 2750 increments a count, and prints the details. Finally, the average salary is printed.

```
(loop for (name salary) being each record in
 [select [ename] [sal] :from [emp]]
 initially (format t "~&~20A~10D" 'name 'salary)
 when (and salary (> salary 2750))
   count salary into salaries
   and sum salary into total
   and do (format t "~&~20A~10D" name salary)
 else
   do (format t "~&~20A~10D" name "N/A")
 finally
 (format t "~2&Av Salary: ~10D" (/ total salaries)))
```

15.3.2 Functional Data Definition Language (FDDL)

Functions in the FDDL may be used to change or query the structure of the database.

15.3.2.1 Queries

The following three commands return information about the entries in a database.

list-tables	Macro
list-tables &key database	
Returns the list of table names in <i>database</i> .	
list-attributes	Function
list-attributes table &key database	
Returns the list of attributes of <i>table</i> .	
attribute-type	Function
attribute-type attribute table &key database	
Returns the type of <i>attribute</i> of <i>table</i> . This is a keyword represen vendor-specific type value.	ting a

15.3.2.2 FDDL Querying example

This example shows you how to query the type of the ename attribute of the emp table.

(attribute-type [ename] [emp]) -> (CHAR)

15.3.2.3 Modification

You may create or drop tables, indexes or views using the following functions.

create-table

Function

create-table name description &key database

Creates a table called *name* and defines its columns and other properties with *description*. The *description* argument is a list of lists of attribute names followed by type information. For example:

is equivalent to the following SQL:

```
CREATE TABLE MANAGER
(ID CHAR(10) NOT NULL,SALARY INTEGER)
```

drop-table

Function

Function

drop-table name &key database

Deletes table *name* from *database*.

create-index

create-index name &key on unique attributes database

Creates an index called *name* on table *on* using *attributes*. For example:

```
(create-index [manager]
    :on [emp]
    :unique t
    :attributes '([ename] [sal]))
```

drop-index

Function

drop-index name &key database

Deletes index *name* from *database*.

create-view

Function

create-view name &key column-list as with-check-option database

Creates a view called *name* using the query *as* and the optional *column*-*list* and *with-check-option*.

This example creates the view manager with the records in the employee table whose job title is MANAGER.

drop-view

Function

drop-view name &key database

Deletes view name from database.

15.4 Object oriented interface

This section describes the object-oriented interface to SQL databases using specialized CLOS classes. These classes have standard-db-object as one of their superclasses and have standard-db-class as their metaclass. The standard-db-class metaclass provides the specialized behavior for mapping subclasses of standard-db-object onto records in the database. A class of this kind is created using def-view-class.

15.4.1 Object oriented/relational model

In the simple case, a class maps onto a database table, an instance of the class maps onto a record in the table, and a slot in the class maps onto an attribute in the table. In general, however, a class maps onto a database view, an instance of the class maps onto a collection of records in the view, and a slot in the class is either:

- A *base slot* that maps onto an attribute in the view
- A join slot that points to a list of other view-class instances

If an instance maps onto more than one record in the view then for each record, all the key attributes from each table in the view are the same.

15.4.2 Object-Oriented Data Definition Language (OODDL)

The OODDL lets you define a mapping between the relational and object-oriented worlds to be defined. Through the mapping a CLOS object can effectively denote a collection of records in a database view, and can contain pointers to other view-based CLOS objects. The CLOS object makes explicit an object implicitly described by the flat relational values.

def-view-class

Macro

Function

def-view-class name superclasses slots &rest class-options

Extends the syntax of defclass to allow special *base slots* to be mapped onto the attributes of database views (presently single tables). When a select query that names a view-class is submitted, then the corresponding database view is queried, and the slots in the resulting instances are filled with attribute values from the database.

It is also possible to create join slots and virtual (ordinary) slots.

All the special slots are distinguished by a modified set of class and slot options. The special slots and their options are described in more detail under def-view-class in the *LispWorks Reference Manual*.

create-view-from-class

create-view-from-class class &key database

Creates a view in *database* based on *class* which defines the view.

drop-view-from-class

Function

```
drop-view-from-class class &key database
```

Drops a view from *database* based on the *class* which defines that view.

15.4.2.1 Example view-class definition

The following example shows a class corresponding to the traditional employees table, with the employee's department given by a join with the departments table. See def-view-class in the *LispWorks Reference Manual* for a description of the slot options.

```
(def-view-class employee (standard-db-object)
   ((employee-number :db-kind :key
                     :column empno
                     :type integer)
    (employee-name :db-kind :base
                   :column ename
                   :type (string 20)
                   :accessor employee-name)
    (employee-department :db-kind :base
                         :column deptno
                         :type integer
                         :accessor employee-department)
    (employee-job :db-kind :base
                  :column job
                  :type (string 9))
    (employee-manager :db-kind :base
                      :column mgr
                      :type integer)
    (employee-location :db-kind :join
                       :db-info (:join-class department
                                 :retrieval :deferred
                                 :set nil
                                 :home-key employee-department
                                 :foreign-key department-number
                                 :target-slot department-loc)
                       :accessor employee-location))
                (:base-table emp))
```

The def-view-class macro allows elements or lists of elements to follow :home-key and :foreign-key. The elements can be symbols, nil, strings, integers or floats.

This syntax means that an object from the join class is only included in the join slot if the values from the home-key are equal to the values in the foreign-key, in order. These values are calculated as follows:

- If the element in the list is a symbol it is taken to be a slot name and the value of the slot is used
- Otherwise the element is taken to be the value

Note that some vendors may have short maximum identifier lengths. The CLOS interface uses constructed alias names for tables in its SQL queries, and long table names or long class names may cause the constructed aliases to exceed the maximum identifier length for a particular vendor.

15.4.3 Object-Oriented Data Manipulation Language (OODML)

The OODML is designed to be powerful and expressive, while remaining familiar to users of the FDML. To achieve this aim, some of the functions and macros in the SQL interface have been *overloaded* — particularly the select function and the iteration constructs.

list-classes

Function

list-classes &key database

The list-classes function returns a list of the class objects of the CLOS classes which have been defined by def-view-class for *database*.

select

Function

select selections &rest args &key all set-operation distinct from where group-by having order-by database

The select function is common across the both the functional and object-oriented SQL interfaces. If *selections* refers to a view class by supplying its symbolic name then the select operation becomes object-oriented — returning a list of instances instead of a list of attributes. The symbol slot-value is a valid SQL operator for use within the *where* clause.

15.4.3.1 Examples

In the following generic functions, *instance* is an instance of a def-view-class representing a record in a database.

update-records-from-instance

Generic function

```
update-records-from-instance instance &key database
```

Updates the record in *database* represented by *instance*. If the instance is already associated with a database, that database is used and *database* is ignored. If *instance* is not yet associated with a database, a record is created for *instance* in the appropriate table of *database* and the instance becomes associated with that database.

update-record-from-slot

update-record-from-slots

Generic function

Generic function

update-record-from-slot instance slot &key database update-record-from-slots instance slots &key database

Updates the individual data item from *slot* or list of *slots* for *instance*. As with update-records-from-instance, *database* is only used if *instance* is not yet associated with a database, in which case a record is created in *database*. Only *slot* or *slots* are initialized in this case; other columns in the underlying database receive default values. Slots are the CLOS slot names; the corresponding column names are derived from the view class definition.

delete-instance-records

Generic function

delete-instance-records instance

Deletes the records represented by *instance* from the database associated with it. If *instance* has no associated database, delete-instance-records signals an error.

15.4.3.2 Iteration

The object-oriented SQL interface has the same three iteration constructs as the functional interface (see Section 15.3.1.4 on page 134): a do-loop, a mapping function, and an extension to the Common Lisp loop macro. However, in this case, the iteration focus is not a tuple of attributes (that is, a record), but a tuple of instances. For example:

```
(loop for (jones company) being the tuples in
    [select 'person 'organisation
    :where [= [slot-value 'person 'surname] "Jones"]]
    do (format t "~A ~A ~%"
                         (slot-value jones 'forename)
                         (slot-value company 'short-name))))
```

Note: Instances may denote many database records, and hence the effective iteration focus in this case is a tuple of sets of tuples of attributes.

15.5 Symbolic SQL syntax

Common SQL supports a symbolic query syntax across both the functional and object-oriented interface layers. It allows SQL and Common Lisp expressions to be mixed together — with as much processing as possible done at compile-time. Symbolic SQL expressions are read as square-bracketed lists to distinguish them from Lisp expressions. However, each can be nested within the other to achieve the desired result.

By default, this reader syntax is turned off. To turn it on see Section 15.5.3 on page 149.

15.5.1 The "[...]" Syntax

The square bracket syntax for the SQL interface is heavily overloaded to provide the most intuitive behavior in all situations. There are three uses of square brackets:

- 1. To enclose a database identifier
- 2. To construct an SQL string representing a symbolic expression
- 3. To enclose literal SQL

Each of these uses is demonstrated below.

15.5.1.1 Enclosing database identifiers

Database identifiers can be enclosed in the square bracket syntax as shown in the following examples.

```
[foo] => #<SQL-IDENT "FOO">
    This case corresponds to an unqualified SQL identifier
    as in: SELECT FOO FROM BAR.
[foo bar] => #<SQL-IDENT "FOO.BAR">
    This corresponds to a qualified SQL identifier as in:
        SELECT FOO.BAR FROM FOO
["foo" bar] => #<SQL-IDENT "\"foo\".BAR">
    This corresponds to a qualified SQL identifier with an
    aliased table name containing special characters as in:
        SELECT "foo".BAR FROM BAZ "foo".
[foo "bar"] => #<SQL-IDENT FOO \"bar\">
    This corresponds to an alias definition as in:
        SELECT "bar".* FROM FOO "bar".
[foo :integer] => #<SQL-IDENT "FOO" :INTEGER>
        As above, but including a type coercion component.
```

[foo bar :integer] -> #<SQL-IDENT "FOO.BAR" :INTEGER>
As above, but includes a type coercion component.

["foo" bar :integer] —> #<SQL-IDENT "\"foo\".BAR" :INTEGER> As above, but includes a type coercion component.

15.5.1.2 SQL strings representing symbolic expressions

There are some SQL operators which may take a single argument (for example any, all, not, union, intersection, minus, group-by, and having). These are read as calls to the appropriate SQL operator. For example:

[any '(3 4)] -> #<SQL-VALUE-EXP "(ANY (3,4))">

This causes no conflict, however, as it is illegal to use these reserved words as identifiers in SQL. Similarly with two argument operators:

The select statement itself may be prepared for later query execution using the [] syntax. For example:

[select [person_id] [surname] :from [person]]

This form results in an SQL expression, which could be bound to a Lisp variable and later given to query to execute. For example:

Strings can be inserted in place of database identifiers within a select:

Any non-constant included gets filled in at runtime, for example:

[> [foo] x]

when macroexpanded reads as

```
(SQL-> #<SQL-IDENT "FOO"> X)
```

which constructs the actual SQL string at runtime.

Any arguments to an SQL operator that are Lisp constants are translated to the matching SQL construct at compile-time, for example:

```
"foo" -> "'foo'"
3 -> "3"
'("this" 5 "that") -> "('this', 5, 'that')"
'xyz -> "XYZ"
```

Other SQL operators which are supported are null, exists, *, +, /, -, like, and, or, in, ||, =, <, >=, <=, count, max, min, avg, sum, distinct, slot-value, and between. The general syntax is: [<operator> <operand> ...], for instance:

(select [count [*]] :from [emp])

15.5.1.3 Enclosing literal SQL

Literal SQL statements can simply be enclosed in the square bracket syntax, as shown below.

```
["SELECT FOO, BAR FROM BAZ"]
-> #<SQL "SELECT FOO, BAR FROM BAZ">
[select [*] :from [tbl]]
-> #<SQL-QUERY "(SELECT * FROM TABLE)">
```

[person surname]
->#<SQL-IDENT "PERSON.SURNAME">
[> [foo] 37]
->#<SQL-RELATIONAL-EXP "(FOO > 37)">

15.5.2 Programmatic interface

In some cases it is necessary to build SQL-expressions dynamically under program control. The following functions are provided to this end:

sql-operation

Function

sql-operation op &rest args

Returns the SQL for an operator applied to its arguments. That is, this is equivalent to the second use of the [] syntax above where SQL strings represent symbolic expressions. This function is shorthand for

(apply (sql-operator op) args)

sql-expression

sql-expression &key string table alias attribute type

Makes an SQL expression from the given keywords. This is equivalent to the first and third uses of the [] syntax as discussed in Section 15.5.1 on page 144. Valid combinations are:

string table table and alias table and attribute table, attribute, and type table-alias and attribute table-alias, attribute, and type attribute attribute Function

sql-operator

Function

```
sql-operator symbol
```

Returns the symbol for an SQL operator (that is, sql-symbol).

sql

Function

sql &rest args

Makes SQL out of *args*. Each argument to sql is turned into SQL and then the *args* are concatenated with a single space between each pair. The rules for translation into SQL are as follows (based on the type of each individual argument x):

string -> (format nil "'~A'" x)

That is, the characters of x between single quotes (this corresponds to an SQL string constant);

(sql null) -> "NULL"

That is, an SQL null value;

```
symbol -> (symbol-name x)
number -> (princ-to-string x)
list -> (format nil "(~{~A~^,~})" (mapcar #'sql x))
```

That is, the elements of ${\bf x}$ in SQL, between parentheses separated by commas.

```
vector -> (format nil "~{~A~^,~}" (map 'list #'sql x))
```

That is, the elements of x in SQL, comma-separated, without parentheses. This is to allow the easy generation of SQL lists that require no parentheses such as table lists in select statements.

```
sql-expression -> x
otherwise -> (error)
```

15.5.2.1 Examples

The following example function, taken from the object-oriented SQL interface layer, makes an SQL query fragment that finds the records corresponding a

CLOS object (using the slots as attributes), when built into the *where*-clause of an updating form.

```
(let* ((class (class-of object))
          (key-slots (db-class-keyfields class)))
   (loop
     for key in key-slots
     for slot-name = (slot-definition-name key)
     for slot-type = (db-slot-definition-type key)
     collect
     [= (make-field-name class key)
        (lisp-to-sql-format
           (slot-value object slot-name)
           (if (listp slot-type)
               (car slot-type)
               slot-type))]
     into cols
     finally (apply (sql-operator 'and) cols)))
->
#<SQL-RELATIONAL-EXP "(EMP.EMPNO = 7369">
```

Here is another example that produces an SQL select statement:

```
(sql-operation 'select
    (sql-expression :table 'foo
                    :attribute 'bar)
    (sql-expression :attribute 'baz)
  :from (list
          (sql-expression :table 'foo)
          (sql-expression :table 'quux))
  :where (sql-operation 'or
            (sql-operation '>
               (sql-expression :attribute 'baz)
             3)
            (sql-operation 'like
               (sql-expression :table 'foo
                               :attribute 'bar)
             "SU%")))
->
#<SQL-QUERY "SELECT FOO.BAR, BAZ FROM FOO, QUUX
  WHERE ((BAZ > 3) OR (FOO.BAR LIKE 'SU%'))">
```

15.5.3 Utilities

The following functions enable and disable the square bracket syntax:

enable-sql-reader-syntax

enable-sql-reader-syntax

Turns on square bracket syntax and sets the state so that restore-sqlreader-syntax-state enables the syntax again if it is subsequently disabled.

disable-sql-reader-syntax

disable-sql-reader-syntax

Turns off square bracket syntax and sets the state so that restore-sqlreader-syntax-state disables the syntax again if it is subsequently enabled.

locally-enable-sql-reader-syntax

Function

Function

Function

Function

Function

locally-enable-sql-reader-syntax

Turns on square bracket syntax and does not change the syntax state — so restore-sql-reader-syntax-state restores the current enable/disable state.

locally-disable-sql-reader-syntax

```
locally-disable-sql-reader-syntax
```

Turns off square bracket syntax and does not change the syntax state — so restore-sql-reader-syntax-state restores the current enable/disable state.

restore-sql-reader-syntax-state

restore-sql-reader-syntax-state

Sets the enable/disable state of the SQL reader syntax to reflect the last call to either enable-sql-reader-syntax or disable-sql-reader-syntax.

Function

Function

```
The intended use of locally-enable-sql-reader-syntax and locally-
disable-sql-reader-syntax is in a file:
```

15.6 SQL I/O recording

It is sometimes convenient to simply monitor the flow of commands to, and results from, a database. A number of functions are provided for this purpose.

The functions operate on two stream collections (*broadcast streams*) — one each for commands and results. They allow the recording to be started and stopped, checked, or recorded on further individual streams.

start-sql-recording	Function
start-sql-recording &key type database	

Starts recording SQL command or result traffic onto a broadcast stream. Initially the broadcast stream for commands or results is just *standardoutput*. Returns no values.

```
stop-sql-recording
```

stop-sql-recording &key type database

Stops recording SQL command or result traffic. Returns no values.

sql-recording-p

```
sql-recording-p &key type database
```

Returns t if SQL command or result traffic is being recorded, otherwise returns mil.

list-sql-streams

list-sql-streams &key type database

Returns the individual streams recording SQL command or result traffic, since there may be multiple streams wrapped up into a single broadcast stream.

sql-stream

sql-stream &key type database

Returns the broadcast stream used for recording SQL command or result traffic.

add-sql-stream

add-sql-stream stream &key type database

Adds a new stream to the broadcast stream for SQL command or result traffic. Returns the new stream.

delete-sql-stream

delete-sql-stream stream &key type database

Deletes a stream from the broadcast stream for SQL command or result traffic. Returns the deleted stream.

15.7 SQL errors

All errors generated by the SQL interface are of type sql-error. Subtypes of sql-error are:

sql-reader-error

Generated by the symbolic SQL interface.

sql-database-error

Generated when the underlying database signals an error.

Function

Function

Function

Function

sql-type-coercion-error

Generated when type coercion of the data from the underlying database causes an error.

sql-type-check-error

Generated when type checking in the SQL interface causes an error.

User Defined Streams

16.1 Introduction

A number of classes and functions are provided in the stream package that allow you to define your own input and output streams. The user defined streams can use the standard I/O functions, and methods specializing on the relevant user defined stream class can also be defined to provide specific implementations of other I/O functions. Note that some changes have been made to the standard I/O functions to allow for this. For example, streamelement-type is now a generic function. See the common-lisp package reference entries in the *LispWorks Reference Manual* for alterations to standard functions, and the stream package entries for more details on user defined streams.

16.2 An illustrative example of user defined streams

In this chapter an example is provided to illustrate the main features of the stream package. In this example a stream class is defined to provide a wrapper for file-stream which uses the Unicode Line Separator instead of the usual ASCII CR/LF combination to mark the end of lines in the file. Methods are then defined, specializing on the user defined stream class to ensure that it handles reading from and writing to a file correctly.

16.2.1 Defining a new stream class

Streams can be capable of input or output (or both), and may deal with characters or with binary elements. The stream package provides a number of stream classes with different capabilities from which user defined streams can inherit. In our example the stream must be capable of input and output, and must read characters. The following code defines our stream class appropriately:

```
(defclass unicode-ls-stream
      (stream:fundamental-character-input-stream
      stream:fundamental-character-output-stream)
  ((file-stream :initform nil
      :initarg :file-stream
      :accessor ls-stream-file-stream)))
```

The new class, unicode-ls-stream, has fundamental-character-inputstream and fundamental-character-output-stream as its superclasses, which means it inherits the relevant default character I/O methods. We shall be overriding some of these with more relevant and efficient implementations later.

Note that we have also provided a slot, called file-stream. This slot is a place holder for a Common Lisp file stream. When making an instance of unicodels-stream we can create an instance of a file stream in this slot. This allows us to use the Common Lisp file stream functionality for reading from and writing to a file.

fundamental-character-input-stream	Class
fundamental-character-output-stream	Class
Package: stream	
The classes fundamental-character-input-stream and fundamer character-output-stream provide default methods for generic fu tions used for character input and character output respectively, a	unc-

should therefore be included by stream classes concerned with these. The user can provide methods for these generic functions specialized on the user-defined classes.

16.2.2 Recognizing the stream element type

We know that the stream will read from a file using file-stream functionality and that the stream element type will be simple-char. The following defines a method on stream-element-type to return the correct element type.

```
(defmethod stream-element-type ((stream unicode-ls-stream))
    'simple-char)
```

stream-element-type

Generic Function

stream-element-type stream

Package: common-lisp

The function stream-element-type is implemented as a generic function. Depending on the stream, a method should be defined for this generic function that takes a *stream* as its argument and returns the element type of the stream.

16.2.3 Stream directionality

Streams can be defined for input only, output only, or both. In our example, the unicode-ls-stream class needs to be able to read from a file and write to a file, and we therefore defined it to inherit from an input and an output stream class. We could have defined disjoint classes instead, one inheriting from fundamental-character-input-stream and the other from fundamental-character-output-stream. This would have allowed us to rely on the default methods for the direction predicates. However, given that we have defined one bi-directional stream class, we must define our own methods for the direction predicates.

```
(defmethod input-stream-p ((stream unicode-ls-stream))
  (input-stream-p (ls-stream-file-stream stream)))
(defmethod output-stream-p ((stream unicode-ls-stream))
  (output-stream-p (ls-stream-file-stream stream)))
```

The above code allows us to "trampoline" the correct direction predicate functionality from file-stream, using the ls-stream-file-stream accessor we defined previously.

input-stream-p

Generic Function

output-stream-p

Generic Function

input-stream-p *stream*

output-stream-p stream

Package: common-lisp

The predicates input-stream-p and output-stream-p are implemented as generic functions. Their default methods return t if *stream* is respectively an input or output stream. If the user wants to implement a stream with no inherent directionality (and thus does not include fundamentalinput-stream or fundamental-output-stream) but for which the directionality depends on the instance, then suitable methods should be provided.

16.2.4 Stream input

The following definition for the stream-read-char reads a character from the stream. If the character read is a #\Line-Separator, then the method returns #\Newline, otherwise the character read is returned. It also returns :eof at the end of the file.

There is no need to define a new method for stream-read-line as the default method uses stream-read-char repeatedly to read a line, and our implementation of stream-read-char ensures that this will work.

stream-read-char Generic Function
stream-read-char stream

Package: stream

The generic function stream-read-char reads one item from stream. The item read is either a character or the end of file symbol :eof if the stream is at the end of a file. Every subclass of fundamental-character-input-stream must define a method for this function.

We also need to make sure that if a #\Newline is unread, it is unread as a #\Line-Separator. The following code uses the Common Lisp file stream function unread-char to achieve this.

stream-unread-char

Generic Function

stream-unread-char stream

Package: stream

The generic function stream-unread-char undoes the last call to stream-read-char, as in unread-char. Every subclass of fundamentalcharacter-input-stream must define a method for this function.

Finally, although the default methods for stream-listen and stream-clearinput would work for our stream, it is faster to use the functions provided by file-stream, using our accessor, ls-stream-file-stream.

```
(defmethod stream:stream-listen ((stream unicode-ls-stream))
  (listen (ls-stream-file-stream stream)))
(defmethod stream:stream-clear-input ((stream unicode-ls-stream))
  (clear-input (ls-stream-file-stream stream)))
```

stream-clear-input

Generic Function

stream-clear-input stream

Package: stream

The generic function stream-clear-input implements clear-input. The default method is defined on fundamental-input-stream and does nothing.

stream-listen

Generic Function

stream-listen stream

Package: stream

The generic function stream-listen is used by listen and returns t if there is input available. The default method uses stream-read-charno-hang and stream-unread-char. Most streams should define their own method as this is usually trivial and more efficient than the method provided.

16.2.5 Stream output

The following definition for the stream-write-char uses write-char to write a character to the stream. If the character written to unicode-ls-stream is a #\Newline, then the method writes a #\Line-Separator to the file stream.

stream-write-char

Generic Function

stream-write-char stream character

Package: stream

The generic function stream-write-char writes character to stream. Every subclass of fundamental-character-output-stream must have a method defined for this function.

The default method for stream-write-string calls the above generic function and successfully write a string to the stream. However, the following is a more efficient implementation for our stream.

We do not need to define our own method for stream-terpri, as the default uses stream-write-char, and therefore works appropriately

To be useful, the stream-line-column and stream-line-start-p generic functions need to know the number of characters preceding a #\Lineseparator. However, since the LispWorks file stream records line position only by #\Newline characters, this information is not available. Hence we define the two generic functions to return nil:

```
(defmethod stream:stream-line-column
 ((stream unicode-ls-stream))
  nil)
(defmethod stream:stream-start-line-p
 ((stream unicode-ls-stream))
  nil)
```

stream-line-column

Generic Function

stream-line-column stream

Package: stream

The generic function stream-line-column returns the column number where the next character will be written from *stream*, or nil if this is not meaningful for the stream. This function is used in the implementation of print and the format ~t directive. A method for this function must be defined for every character output stream class that is defined, although at its simplest it may be defined to always return nil.

stream-start-line-p

Generic Function

stream-start-line-p stream

Package: stream

The generic function stream-start-line-p returns t if stream is positioned at the beginning of a line, and nil otherwise. It is permissible to define a method that always returns nil.

Finally, the methods for stream-force-output, stream-finish-output and stream-clear-output are "trampolined" from the standard force-output, finish-output and clear-output functions.

16.2.6 Instantiating the stream

Now that the stream class has been defined, and all the methods relevant to it have been set up, we can create an instance of our user defined stream to test it. The following function takes a filename and optionally a stream direction as its arguments and makes an instance of unicode-ls-stream. It ensures that the file-stream slot of the stream contains a Common Lisp file-stream capable of reading from or writing to a file given by the filename argument.

```
(defun open-unicode-ls-file (filename &key (direction :input))
 (make-instance 'unicode-ls-stream :file-stream
    (open filename
       :direction direction
       :external-format :unicode
       :element-type 'simple-char)))
```

The following macro uses open-unicode-ls-stream in a similar manner to the Common Lisp macro with-open-file:

We now have the required functions and macros to test our user defined stream. The following code uses config.sys as a source of input to an instance of our stream, and outputs it to the file unicode-ls.out, changing all occurrences of #\Newline to #\Line-Separator in the process.

After running the above code, if your load the file C:\unicode-ls.out into an editor (for example, a LispWorks editor), you can see the line separator used instead of CR/LF. Most editors do not yet recognize the Unicode Line Separator character yet. In some editors it appears as a blank glyph, whereas in the LispWorks editor it appears as <2028>. In LispWorks you can use Alt+x what Cursor Position or Ctrl+x = to identify the unprintable characters.

You can also use the follow code to print out the contents of the new file line by line.

```
(with-open-unicode-ls-file (ss "C:\\unicode-ls.out")
  (loop while (when-let (line (read-line ss nil nil))
                (write-line line))))
```

User Defined Streams

Index

Symbols

! 85 :< debugger command 10 \$(dollar) variable 20 \$\$ variable 20 \$\$\$ variable 20 :> debugger command 10 [...] syntax in SQL 144

A

:a debugger command 14 abort restart 7 action lists 51 defining 52 examples 56 undefining 52 adding actions to action lists 54 add-special-free-action function 76 add-sql-stream function 152 add-symbol-profiler function 78 advice after 43 around 43 before 43 example of use 47 facility 41 for macros 45 for methods 45 main chapter 41 removing 44 after advice 43 :after keyword 30 :all debugger command 11 :all keyword 96 all, SQL operator 145

:allocation keyword 34
allocation-in-gen-num function 75
allocation-in-gen-num macro 72
any, SQL operator 145
around advice 43
attribute-type function 136
avoid-gc function 76

B

:b debugger command 9
backtrace 8
 quick backtrace 9
 verbose backtrace 9
:backtrace keyword 31
base slot 139
before advice 43
:before keyword 29
:bindings keyword 18
:bq debugger command 10
:break keyword 31
:break-on-exit keyword 31

С

:c debugger command 14 call-next-advice function 44 catch frame, examining 8 :catchers keyword 18 :caused-by keyword 96 :cc debugger command 11 classes database 125 standard-db-class 138 standard-db-class 138 clean-down function 75 clean-generation-0 function 75 collect-generation-2 function 75 collect-highest-generation function 75 commit function 131 commitfunction 133 Common Lisp systems. See system compiler comparison with interpreter 59 control 62 levels of safety 63 main chapter 59 optimization of 62-65 workings of 61 compile-system function 94 connect function 127 connected-databases function 127 *connect-if-exists* variable 127 continue restart 7 create-index function 137 create-process function 88 create-table function 137 create-view function 138 create-view-from-class function 139 creation of process 88 current frame 10 current process 88 *current-process* variable 88 customization main chapter 83 of editor 84

D

database classes in SQL 125 connection in SQL 126 database class 125 database-name function 128 DDL 136, 139 debugger commands 8 control variables 16 invoking from the tracer 31 main chapter 5 debugger commands :< 10 :> 10 :a 14 :all 11 :ь 9 :bg 10 :c 14

:cc 11 :ed 11 :error 11 :qet 12 :his 13 :lambda 12 :1f 13 :n 10 :p 10 :redo 12 :res 14 :ret 14 :top 14 :use 12 :v 11 *debug-io* variable 16 *debug-print-length* variable 17 *debug-print-level* variable 17 declaim macro 67 declare special form 62, 66 defadvice macro 42, 45, 50 *default-database* variable 125, 126 *default-database-type* variable 126 :default-pathname keyword 95 define-action macro 54 define-action-list macro 52 definition specs 37 defparser error handling with 104 defparser macro 101 functions defined by 103 defsystem macro 95, 95-99 examples of use 98 def-view-class macro 138, 139 delete-advice macro 44, 50 delete-instance-records generic function 143 delete-records function 131 delete-recordsfunction 132 delete-sql-stream function 152 describe function 19 *describe-length* variable 21 *describe-level* special variable 20 describe-object generic function 20 *describe-print-length* variable 20 *describe-print-level* variable 20 diagnostic utilities for action lists 55 disable-sql-reader-syntax function 150 disconnect function 128 DML 129.141

\$(dollar) variable 20
\$\$ variable 20
\$\$\$ variable 20
:dont-know keyword 11
do-query macro 134
drop-index function 138
drop-table function 137
drop-view function 138
drop-view-from-class function 140

Е

:ed debugger command 11 editor customizing 84 enable-sql-reader-syntax function 129. 150 :entrycond keyword 32 :error debugger command 11 error handling in parser generator 104 errors in SQL 152 :eval-after keyword 30 :eval-before keyword 30 evaluating forms during tracing 29-31 exception handling for action lists 54 execute-command function 134 execution functions 51 execution profiling 77 execution stack examining 8 :exitcond keyword 32 expand-generation-1 function 75 extended-time macro 75,81

F

fasl (fast load)
 description 59
FDDL 136
FDML 141
files
 load-on-demand 85
find-database function 128
find-object-size function 75
find-process-from-name function 88
flag-not-special-free-action
 function 76
flag-special-free-action function 76
forms
 evaluating when tracing 29-31

frame, examining 8 free-important-locks function 91 function, altering with advice 41 Functional DDL 136 Functional DML 129 functional interface in SQL 129 functions add-special-free-action 76 add-sql-stream 152 add-symbol-profiler 78 allocation-in-gen-num 75 arguments for traced 29 attribute-type 136 avoid-gc 76 call-next-advice 44 clean-down 75 clean-generation-0 75 collect-generation-2 75 collect-highest-generation 75 commit 131,133 compile-system 94 connect 127 connected-databases 127 create-index 137 create-process 88 create-table 137 create-view 138 create-view-from-class 139 database-name 128 delete-records 131,132 delete-sql-stream 152 describe 19 disable-sql-reader-syntax 150 disconnect 128 drop-index 138 drop-table 137 drop-view 138 drop-view-from-class 140 enable-sql-reader-syntax 129, 150 execute-command 134 expand-generation-1 75 find-database 128 find-object-size 75 find-process-from-name 88 flag-not-special-free-action 76 flag-special-free-action 76 free-important-locks 91 gc-if-needed 76 get-default-generation 75 get-gc-parameters 76 initialize-database-type 125,126 initialize-multiprocessing 89

insert-records 131 inspect 20 list-all-processes 88 list-attributes 136 list-classes 141 list-sql-streams 152 locally-disable-sql-readersyntax 150 locally-enable-sql-readersyntax 150 make-lock 90 map-guery 134 mark-and-sweep 76 normal-gc 76 print-actions 55 print-profile-list 79 print-query 130 process-allow-scheduling 88 process-arrest-reasons 88 process-break 89 process-interrupt 89 process-kill 89 process-lock 91 process-name 88 process-priority 89 process-run-function 88 process-run-reasons 89 process-unlock 91 process-wait 88,89 process-wait-function 89 process-wait-with-timeout 88,89 proclaim 62,66 ps 88 query 134 remove-advice 44,50 remove-special-free-action 76 remove-symbol-profiler 78 reset-profiler 78 restore-sql-reader-syntaxstate 150 results for traced 30 rollback 131, 133 room 75 select 129,141 set-debugger-options 17 set-default-generation 72.75 set-gc-parameters 71, 73, 74, 76 set-minimum-free-space 73,75 set-profiler-threshold 78 set-promotion-count 73,75 set-up-profiler 78 sal 148

sql-expression 147 sql-operation 147 sql-operator 148 sql-recording-p 151 sql-stream 152 start-sql-recording 151 status 128 stop-sql-recording 151 switch-static-allocation 70,71 symbol-to-string 103 toggle-source-debugging 67 total-allocation 75 tracing inside 35 update-records 131,132 without-interrupts 89 without-preemption 89 fundamental-character-input-stream class 156 fundamental-character-output-stream class 156

G

garbage collection, see also storage management main chapter 69 gc-if-needed function 76 generation definition 70 generation 2 75 generic functions delete-instance-records 143 describe-object 20 input-stream-p 158 output-stream-p 158 stream-clear-input 159 stream-element-type 157 stream-line-column 161 stream-listen 160 stream-read-char 158 stream-start-line-p 162 stream-unread-char 159 stream-write-char 160 update-record-from-slot 142 update-record-from-slots 142 update-records-from-instance 142 :get debugger command 12 get-default-generation function 75 get-gc-parameters function 76 grammar non-terminal 102 resolving ambiguities 102 graphics ports 2

>, SQL operator 145 group-by, SQL operator 145

Н

handler frame, examining 8 :handler keyword 18 having, SQL operator 145 :hidden keyword 18 *hidden-packages* variable 17 :his debugger command 13 hook functions 51

I

initialization of SQL 125 initialize-database-type function 125 initialize-database-type function 126 *initialized-database-types* variable 126 initialize-multiprocessing function 89 *initial-processes* variable 88 input-stream-p generic function 158 insert-records function 131 :inside keyword 35 inspect function 20 inspector main chapter 19 teletype 19 *inspect-print-length* variable 21 *inspect-print-level* variable 21 in-static-area macro 71 interface between parser generator and lexical analyser 105 SQL initialization 125 interpreter differences from compiler 59 intersection, SQL operator 145 introduction main chapter 1 :invisible keyword 18

J

join slot 139

K

keywords :after 30 :all 96 :allocation 34 :backtrace 31 :before 29 :bindings 18 :break 31 :break-on-exit 31 :catchers 18 :caused-by 96 :default-pathname 95 :dont-know 11 :entrycond 32 :eval-after 30 :eval-before 30 :exitcond 32 :handler 18 :hidden 18 :inside 35 :invisible 18 :maximum-buffer-size 71 :maximum-overflow 74 :members 95 :minimum-buffer-size 71 :minimum-for-sweep 73,74 :minimum-overflow 74 :new-generation-size 74 :non-symbol 18 :package 95 :previous 97 :process 34 :requires 97 :restarts 18 :rules 96 :source-only 96 :step 32 :trace-output 33 :when 34

L

:lambda debugger command 12
levels of safety, see compiler
:lf debugger command 13
lightweight processes 87
lightweight processes in LispWorks
 customizing 83
 lightweight processes in 87
 processes 87
list-all-processes function 88
list-attributes function 136
list-classes function 141
list-sql-streams function 152
list-tables macro 136

load-on-demand 85 locally-disable-sql-reader-syntax function 150 locally-enable-sql-reader-syntax function 150 locks 90 loop macro 135 extensions in Common SQL 143 loop, extensions in Common SQL 135

Μ

macros advice 45 allocation-in-gen-num 72 declaim 67 defadvice 42, 45, 50 define-action 54 define-action-list 52 defparser 101,103 defsystem 95,95-99 def-view-class 138,139 delete-advice 44,50 do-query 134 extended-time 75,81 in-static-area 71 list-tables 136 loop 135 undefine-action 54undefine-action-list 53 with-heavy-allocation 76 with-lock 91 without-interrupts 76 with-transaction 131,133 make-lock function 90 map-query function 134 mark 72 and sweep 72 mark-and-sweep function 76 :maximum-buffer-size keyword 71 *maximum-ordinary-windows* variable 84 :maximum-overflow keyword 74 *max-trace-indent* variable 37 :members keyword 95 memory allocation during tracing 34 memory management garbage collection strategy 73 image reduction 75 mark and sweep 72 overflow 74 timing in 75 method

advice 45 methods tracing 37 :minimum-buffer-size keyword 71 :minimum-for-sweep keyword 73, 74 :minimum-overflow keyword 74 minus, SQL operator 145 modifying a database 131 multi-processing locks 90

N

:n debugger command 10
:new-generation-size keyword 74
:non-symbol keyword 18
non-terminal in grammar 102
normal-gc function 76
not, SQL operator 145

0

object object-oriented interface in SQL 138 static 71 Object Oriented DDL in SQL 139 Object Oriented DML in SQL 141 OODDL 139 OODML 141 optimization of compiler 62 optimize qualities 63–65 output trace 33 output-stream-p generic function 158

P

:p debugger command 10 package hiding 17 :package keyword 95 parser generator main chapter 101 parser, error handling 104 :previous keyword 97 print-actions function 55 *print-binding-frames* variable 17 *print-catch-frames* variable 17 *print-handler-frames* variable 17 *print-non-symbol-frames* variable 17 *print-open-frames* variable 17 print-profile-list function 79 print-query function 130 *print-restart-frames* variable 17

process creation 88 current 88 in LispWorks 87 :process keyword 34 process-allow-scheduling function 88 process-arrest-reasons function 88 process-break function 89 *process-initial-bindings* variable 88 process-interrupt function 89 process-kill function 89 process-lock function 91 process-name function 88 process-priority function 89 process-run-function function 88 process-run-reasons function 89 process-unlock function 91 process-wait function 88.89 process-wait-function function 89 process-wait-with-timeoutfunction 88, 89 proclaim function 62,66 profiler interpretation of results 79 main chapter 77 pitfalls 80 setting up 78 *profile-symbol-list* variable 78 profiling execution 77 program 77 program profiling 77 ps function 88

Q

query function 134 quick backtrace 9

R

redo 85
:redo debugger command 12
remove-advice function 44, 50
remove-special-free-action
 function 76
removing actions from action lists 54
:requires keyword 97
:res debugger command 14
reset-profiler function 78
restart 7

restart frame, examining 8
:restarts keyword 18
restore-sql-reader-syntax-state
 function 150
:ret debugger command 14
rollback function 131, 133
room function 75
:rules keyword 96

S

segmentation violation in compiled code 65 select function 141 select function 129 select, SQL operator 145 set-debugger-options function 17 set-default-generation function 72,75 set-gc-parameters function 71, 73, 74, 76 set-minimum-free-space function 73,75 set-profiler-threshold function 78 set-promotion-count function 73,75 set-up-profiler function 78 show paths from Editor command 67 :source-only keyword 96 special forms declare 62,66 special variables *describe-level* 20 SQL [...] syntax 144 database classes 125 database connection 126 errors 152 Functional DDL 136 Functional DML 129 functional interface 129 I/O recording 151 interface initialization 125 iteration 143 main chapter 123 **Object Oriented DDL 139** Object Oriented DML 141 object-oriented interface 138 programmatic interface 147 symbolic syntax 143 transaction handling 132 utilities 149 SQL errors sql-database-error 152 sgl-error 152 sql-reader-error 152 sql-type-check-error 153

sql-type-coercion-error 153 sgl function 148 sql-database-error error 152 sql-error error 152 sql-expression function 147 sql-operation function 147 sql-operator function 148 sql-reader-error error 152 sql-recording-p function 151 sql-stream function 152 sql-type-check-error error 153 sql-type-coercion-error error 153 square bracket syntax 144 stack examining 8 standard-db-class class 138 standard-db-object class 138 *standard-output* variable 130 start-sql-recording function 151 static object allocation in memory management 71 status function 128 :step keyword 32 stepper, entering when tracing 32 stop-sql-recording function 151 storage management main chapter 69 stream-clear-input generic function 159 stream-element-type generic function 157 stream-line-column generic function 161 stream-listen generic function 160 stream-read-char generic function 158 streams defining new 156 directionality 157 example 156-163 input 158 instantiating 162 output 160 user defined 155 stream-start-line-p generic function 162 stream-unread-char generic function 159 stream-write-char generic function 160 sweep 72 switch-static-allocation function 70.71

symbol-alloc-gen-num variable 72, 75 symbolic query syntax 143 symbolic syntax in SQL 143 symbol-to-string function 103 syntax, in SQL 143 system defining 94-99 introduction to 93-94 members of 95 plan 96 rules 96-97

Т

teletype inspector 19 toggle-source-debugging function 67 tools inspector 19 :top debugger command 14 total-allocation function 75 trace main chapter 27 *traced-arglist* variable 29, 30, 32, 38 *traced-results* variable 30, 38 *trace-indent-width* variable 38 *trace-level* variable 38 :trace-output keyword 33 *trace-output* variable 38 *trace-print-circle* variable 38 *trace-print-length* variable 20, 38 *trace-print-level* variable 20, 38 *trace-print-pretty* variable 39 tracer :after option 30 :allocation option 34 :before option 29 :break option 31 :break-on-exit option 31 commands available 29-36 definition specs 37 directing output 33 entering the stepper 32 :entrycond option 32 :eval-after option 30 :eval-before option 30 evaluating forms 29-31 example of use 27 :exitcond option 32 functions, tracing inside 35 information displayed 28 :inside option 35 invoking the debugger 31 memory allocation 34

methods, tracing 37 :process option 34 restricting to a process 34 :step option 32 traced function, arguments for 29 :trace-output option 33 trced functions, results for 30 transaction handling 132

U

undefine-action macro 54 undefine-action-list macro 53 union, SQL operator 145 update-record-from-slot generic function 142 update-record-from-slots generic function 142 update-records function 131, 132 update-records-from-instance generic function 142 :use debugger command 12 user defined stream 155 utilities in SQL 149

V

:v debugger command 11 variables \$ (dollar) 20 \$\$ 20 **\$\$\$** 20 *connect-if-exists* 127 *current-process* 88 *debug-io* 16 *debug-print-length* 17 *debug-print-level* 17 *default-database* 125,126 *default-database-type* 126 *describe-length* 21 *describe-print-length* 20 *describe-print-level* 20 *hidden-packages* 17 *initialized-database-types* 126 *initial-processes* 88 *inspect-print-length* 21 *inspect-print-level* 21 *maximum-ordinary-windows* 84 *max-trace-indent* 37 *print-binding-frames* 17 *print-catch-frames* 17 *print-handler-frames* 17 *print-non-symbol-frames* 17

print-open-frames 17 *print-restart-frames* 17 *process-initial-bindings* 88 *profile-symbol-list* 78 *standard-output* 130 *symbol-alloc-gen-num* 72,75 *traced-arglist* 29, 30, 32, 38 *traced-results* 30,38 *trace-indent-width* 38 *trace-level* 38 *trace-output* 38 *trace-print-circle* 38 *trace-print-length* 20,38 *trace-print-level* 20,38 *trace-print-pretty* 39 for action lists 55 verbose backtrace 9 virtual (ordinary) slots 139

W

:when keyword 34 with-heavy-allocation macro 76 with-lock macro 91 without-interrupts function 89 without-interrupts macro 76 without-preemption function 89 with-transaction macro 131 with-transactionmacro 133