Common LispWorks User Guide

Version 5.1

Copyright and Trademarks

Common LispWorks User Guide (Unix version)

Version 5.1

February 2008

Copyright © 2008 by LispWorks Ltd.

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 LispWorks Ltd.

The information in this publication is provided for information only, is subject to change without notice, and should not be construed as a commitment by LispWorks Ltd. LispWorks Ltd assumes no responsibility or liability for any errors or inaccuracies that may appear 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 and KnowledgeWorks are registered trademarks of LispWorks Ltd.

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.

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.

LispWorks contains part of ICU software obtained from http://source.icu-project.org and which bears the following copyright and permission notice:
ICU License - ICU 1.8.1 and later
COPYRIGHT AND PERMISSION NOTICE
Copyright © 1995-2006 International Business Machines Corporation and others. All rights reserved.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR 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.
Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder. All trademarks and registered trademarks mentioned herein are the property of their respective owners.

US Government Restricted Rights
The LispWorks Software is a commercial computer software program developed at private expense and is provided with restricted rights. The LispWorks Software may not be used, reproduced, or disclosed by the Government except as set forth in the accompanying End User License Agreement and as provided in DFARS 227.7202-1(a), 227.7202-3(a) (1995), FAR 12.212(a)(1995), FAR 52.227-19, and/or FAR 52.227-14 Alt III, as applicable. Rights reserved under the copyright laws of the United States.

Address

LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839 (toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 305 468 5262

From elsewhere: +44 870 2206189

www.lispworks.com

Contents

Preface xi

1 Introduction 1

Major tools 2

2 A Short Tutorial 5

Starting the environment 6
Creating a Listener 7
Using the Debugger 9
Viewing output 11
Inspecting objects using the Inspector 12
Examining classes in the Class Browser 14
Summary 17

3 Common Features 19

Displaying tool windows 20
Setting global preferences 25
Performing editing functions 28
The history list 30
Operating on files 31
Displaying packages 32
Performing operations on selected objects 35
Using different views 37
Tracing symbols from tools 40

Linking tools together 41
Filtering information 42
Regexp matching 44
Completion 46
Examining a window 49

4 Getting Help 51

Online manuals in HTML format 51
Online help for editor commands 55
Browsing manuals online using Adobe Reader 55
Reporting bugs 56

5 Manipulating Graphs 57

An overview of graphs 57
Searching graphs 58
Expanding and collapsing graphs 59
Moving nodes in graphs 60
Displaying plans of graphs 61
Preferences for graphs 63
Using graphs in your programs 68

6 The Podium 69

The podium window 69
Specifying the initial tools 70

7 The Class Browser 71

Simple use of the Class Browser 72
Examining slot information 79
Examining superclasses and subclasses 81
Examining classes graphically 84
Examining generic functions and methods 88
Examining initargs 91
Examining class precedences 93

8 The Object Clipboard 97

Placing objects on the Object Clipboard 98
Browsing clipped objects 101

Removing objects 102
Filtering 102
Using the Object Clipboard with a Listener 103

9 The Compilation Conditions Browser 107

Introduction 107
Examining error conditions 108
Configuring the display 109
Access to other tools 113

10 The Debugger Tool 115

Description of the Debugger 117
What the Debugger tool does 122
Simple use of the Debugger tool 123
The stack in the Debugger 124
An example debugging session 124
Performing operations on the error condition 127
Performing operations on stack frames 127
Performing operations on frame variables 128
Configuring the debugger tool 129

11 The Tracer 133

Introduction 133
Tracing and Untracing functions 133
Examining the output of tracing 134
Example 135

12 The Editor 141

Displaying and editing files 143
Displaying output messages in the Editor 146
Displaying and swapping between buffers 146
Displaying Common Lisp definitions 150
Changed definitions 151
Finding definitions 154
Setting Editor preferences 155
Basic Editor commands 164
Other essential commands 169

Cutting, copying and pasting using the clipboard 170
Cutting, copying and pasting using the kill ring 171
Searching and replacing text 174
Using Lisp-specific commands 178
Help with editing 185

13 The Function Call Browser 187

Introduction 187
Examining functions using the graph views 188
Examining functions using the text view 191
Configuring the function call browser 193
Configuring graph displays 194
Performing operations on functions 195

14 The Generic Function Browser 197

Examining information about methods 198
Examining information about combined methods 201
Configuring the Generic Function Browser 206

15 The Search Files tool 207

Introduction 207
Performing searches 209
Viewing the results 215
Configuring the Search Files tool 216

16 The Inspector 223

Inspecting the current object 223
Description of the Inspector tool 224
Filtering the display 225
Examining objects 227
Operating upon objects and items 229
Configuring the Inspector 234
Customizing the Inspector 237
Creating new inspection formats 237

17 The Symbol Browser 243

Introduction 243

Description of the Symbol Browser 244
Configuring the Symbol Browser 248

18 The Interface Builder 249

Description of the Interface Builder 250
Creating or loading interfaces 251
Creating an interface layout 254
Creating a menu system 258
Editing and saving code 262
Performing operations on objects 265
Performing operations on the current interface 269
Performing operations on elements 271

19 Example: Using The Interface Builder 273

Creating the basic layout 275
Specifying attribute values 277
Creating the menu system 280
Specifying callbacks in the interface definition 282
Saving the interface 284
Defining the callbacks 285
Creating a system 288
Testing the example interface 288

20 The Listener 291

The basic features of a Listener 292
Evaluating simple forms 293
Re-evaluating forms 294
Interrupting evaluation 295
The History menu 295
The Expression menu 295
The Values menu 297
The Debug menu 297
Execute mode 298
Setting Listener preferences 302
Running Editor forms in the Listener 303
Help with editing in the Listener 303

21 The Output Browser 305

Interactive compilation messages 307

22 The Process Browser 311

The process list 313
Process control 314
Other ways of breaking processes 315
Updating the Process Browser 315
Process Browser Preferences 315

23 The Profiler 317

Introduction 317
Display of Profiler Data 320
A description of profiling 323
Steps involved in profiling code 324
Format of the cumulative results 329
Interpreting the cumulative results 329
Profiling pitfalls 330
Some examples 332

24 The Shell and Remote Shell Tools 335

Introduction 335
The Shell tool 335
Command history in the shell 336
Configuring the shell to run 337
The Remote Shell tool 337

25 The Stepper 339

Introduction 339
Simple examples 341
The implementation of the Stepper 345
Stepper controls 346
Stepper restarts 350
Breakpoints 350
Stepping macro forms 355
Listener area 357
Configuring the Stepper 358

26 The System Browser 363

Introduction 363
A brief introduction to systems 364
The System Browser 366
A description of the System Browser 366
Examining the system graph 367
Examining systems in the text view 370
Generating and executing plans in the preview view 372
Examining output in the output view 375
Configuring the display 377
Setting options in the system browser 378

27 The Window Browser 379

Introduction 379
Configuring the Window Browser 382
Performing operations on windows 385

28 The Application Builder 387

Introduction 387
Preparing to build your application 389
Building your application 391
Editing the script 392
Troubleshooting 393
Running the saved application 393
Configuring the Application Builder 395

Index 397

Preface

Conventions used in this manual

This manual assumes that you have at least a basic knowledge of Common Lisp. Many source code examples are used throughout the manual to illustrate important concepts, but only extensions to Common Lisp which are specific to the environment are explained in detail.

This manual *does* provide a complete description of the windowed development environment available in your Lisp image. This includes a description of the user interface itself, and a description of how the user interface interacts with Common Lisp.

Using the mouse

Throughout this manual, actions that you perform using the mouse are described in terms of the gesture used, rather than the combination of mouse buttons and keys that need to be used to perform the operation. This is because the buttons that are used are highly dependent on the platform you are running your Lisp image on, the operating system you are using, and even the type of mouse that you have attached to your computer. The mouse gestures available in the environment are described below.

Select

This is by far the most common mouse gesture, and is used for nearly all mouse operations in the environment. Use the select gesture to

- display a menu,
- choose a command from a menu which is already displayed,

- select items from a list or graph
- select or deselect a toggle switch,
- click on a button,
- position the mouse pointer in a piece of text.

Depending on the characteristics of your operating system or (if you are using a UNIX system) your window manager, you may also need to use select in order to move the mouse focus to another window.

If you are using a mouse with several buttons, you can nearly always select by clicking the left-most button, but you should refer to the documentation for your operating system or window manager if you are unsure. This is particularly true if you are using a mouse which has been set up for use by a left-handed person, since it is possible that the function of the mouse buttons has been reversed.

Multiple select

Multiple selection is used in lists and graphs when you want to select more than one item. You can select several items from any list or graph in the environment, and there are a large number of commands which can operate equally well on these multiple selections.

There are a number of standard ways of making multiple selections in a list or graph, depending on your operating system or window manager. Check the relevant documentation if you are unsure, or try any of the following:

- Holding down the **`Shift`** key while selecting an item.
- Holding down the **`Control`** key while selecting an item.
- The middle mouse button (if you have a three-button mouse).

Typically, in lists, holding down the **`Shift`** key lets you make a contiguous selection, and holding down the **`Control`** key lets you make a discontiguous selection.

- To select a block of items from a list, select the first item, hold down the **`Shift`** key, and then select the last item; the intervening items are also selected.
- To select several items which do not form a block, hold down the **`Control`** key while selecting each item individually.

This behavior is typical in a number of operating systems or window managers. You are probably familiar with it if you are familiar with using a mouse.

Double-click

The double-click gesture consists of two select gestures, performed in rapid succession. In general, any item in a list or graph may be double-clicked.

Double-clicking is usually a shortcut for selecting an item and choosing a common menu command, and the precise action that takes place depends on the context in which the double-click was performed.

Double-clicking can only be performed on single selections.

Alternate select

This is a less common gesture, and is used almost exclusively within Common LispWorks to display a context menu (sometimes referred to as the "context menu" or the "right button menu").

If you are using a mouse with several buttons, you should find that you can perform this gesture by clicking the right-most mouse button. On a Macintosh with a single button mouse, the context menu is raised by holding down the `Control` key and clicking the mouse button. Refer to the documentation for your window manager or operating system if you are unsure.

Choosing menu commands

Throughout this manual, menu commands are shown in **This Bold Font**. In addition, submenus are indicated by use of the **>** character. Thus, for instance, the instruction

> "Choose **File > Open**"

means that you should select the **File** menu on a menu bar, and choose the **Open** command in the menu that appears. Similarly,

> "Choose **Works > Tools > Editor**"

means that you should display the **Works** menu by selecting it, select **Tools** from this menu to display a submenu, and choose the **Editor** command from this submenu.

Using the keyboard

Throughout this manual there are descriptions of commands that you can choose by typing at the keyboard. This is especially true when discussing the built-in editor,

which relies heavily on the use of keyboard commands, and the Common Lisp listener, which uses many of the same commands.

Throughout this manual, keyboard input including the names of keys you press is shown in **`This Font`**.

Keyboard commands generally use a combination of ordinary keys together with the modifier keys **`Control`**, **`Shift`**, **`Escape`**, **`Alt`**, **`Meta`** and **`Command`** (not all of these are available on each platform).

UNIX implementation note: You should use the **`Meta`** or **©** key wherever this manual refers to the **`Alt`** key.

In all cases, the **`Control`**, **`Shift`**, **`Meta`** and **`Command`** keys should be held down concurrently with the specified letter. For example:

> **`Ctrl+S`** is read as "hold down the Control key and press S".
>
> **`Ctrl+Shift+A`** is read as "hold down the Control and Shift keys and press A".

In the editor in EMACS emulation mode, instead using the **`Meta`** (**`Alt`**) modifier with a key, the **`Escape`** key can be pressed and released before pressing the key. For example:

> **`Esc E`** is read as "press and release the Escape key, then press E".
>
> **`Alt+E`** is read as "hold down the Alt key and press E".

The two key inputs above are equivalent in EMACS emulation mode. This manual generally refers to **`Alt`** when referring to the editor key strokes.

For more information on using keyboard commands in the built-in editor and the Listener, see "Using keyboard commands" on page 145.

1

Introduction

This manual gives you a complete guide to the Common LispWorks development environment. This environment comprises a large number of window-based tools which have been designed with the Common Lisp developer in mind. The following are among the features provided by the environment:

- A fully functional code Editor specifically designed to make writing Common Lisp source code as swift as possible, emulating Emacs or KDE/Gnome key styles
- A Common Lisp Listener for evaluating Common Lisp forms interactively.
- A range of debugging tools including a graphical Debugger, source code Stepper, code Profiler, Tracer, and the Inspector.
- A range of browsers for examining different objects in your Lisp image, such as the generic functions or CLOS classes that have been defined.
- A tool for simplifying source code management; vital if you are involved in developing large applications.
- (Microsoft Windows, Linux and FreeBSD platforms only) A tool for designing window-based interfaces to your applications. A point-and-click interface is used to design the interface, and Lisp code is generated for you.

- A Shell window that lets you run system utilities (DOS commands on Windows, shell commands on Unix-based systems) inside LispWorks. Remote shells are also supported on Unix-based systems.
- A Search Files tool that allows you to find text matching a regular expression in files.
- An Object Clipboard that allows you to manage selected and copied objects.
- (X11/Motif IDE only) A Lisp Monitor that runs as a separate process to show real time memory use and management.

Because of the large number of tools available, consistency is a vital theme in the environment; each tool has a similar look and feel so that you need only spend a minimum amount of time learning how to use the environment.

In addition, there is a high degree of integration between the tools available. This means that it is possible to transfer pieces of information throughout the environment in a logical fashion; if you create an object in the Listener, you can examine it by transferring it directly to the Inspector. The class of objects that it belongs to can be examined by transferring it to a Class Browser, and from there, the generic functions which have methods defined on it can be browsed.

To reflect these themes of consistency and integration, the earlier chapters in this manual deal with the generic aspects of the environment, while at the same time introducing you to the more important tools.

1.1 Major tools

The environment supports a wide range of tools which can help you to work on your Lisp source code more quickly and efficiently. This section gives you a brief introduction to the most important tools.

You can create any of the tools described here by choosing the appropriate command from the **Tools** menu of the podium window, or by selecting the relevant tool from the **Works > Tools** menu on any other tool.

For full details about any of these tools, see the relevant chapter. The second part of this manual covers each of the tools in the order that they are found on the **Tools** menu.

1.1.1 The Listener

A Common Lisp Listener is provided to let you evaluate Common Lisp forms. This tool is invaluable as a method of testing your code without necessitating compilation or evaluation of whole files of Common Lisp source code.

1.1.2 The Editor

A built-in editor is provided to allow you to develop Common Lisp code. It is based on EMACS, an editor which you may already be familiar with. As an alternative to EMACS keys, the editor offers KDE/Gnome emulation.

The built-in editor offers a wide range of functions specifically designed to help you develop Common Lisp code, and it is fully integrated into the environment so that code being developed is immediately available for testing.

1.1.3 The Class Browser

This tool allows you to examine the Common Lisp classes that are defined in your environment. You can look at the superclasses and subclasses of a given class and see the relationships between them, and you can examine the slots available for each class.

In addition, you can examine the functions and methods defined on a given class, or the precedence list or initargs for the class.

1.1.4 The Output Browser

The Output Browser collects and displays all output from the environment which may be of use. This includes warning and error messages displayed during compilation and output generated by tracing or profiling functions. Many other tools in the environment also provide you with an output *view*, which lets you see any output which is appropriate to that tool.

1.1.5 The Inspector

The Inspector lets you examine and destructively modify the contents of Common Lisp objects. It is an invaluable tool during development, since it lets you inspect the state of any part of your data at any stage during execution. Thus, it is easy to see the

value of a slot and, if necessary, alter its value, so that you can test out the effects of such an alteration before you make the changes necessary in the source code itself.

1.1.6 The Object Clipboard

The Object Clipboard is used to manage multiple Lisp objects. You can select any object in the Object Clipboard for use in paste operations.

As an example of adding a Lisp object to the Object Clipboard, follow these steps:

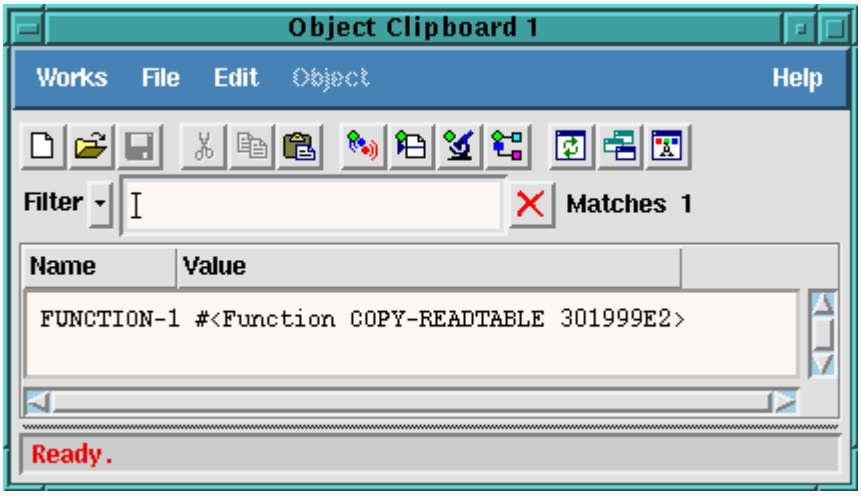
1. Evaluate a Lisp expression in the Listener window. Its value is printed.
2. Choose the menu command **Values > Clip**.

The value from the Listener is now in the Object Clipboard.

If you have not already made an Object Clipboard visible, then do so now using the menu command **Works > Tools > Object Clipboard**.

The Object Clipboard can be seen in Figure 1.1.

Figure 1.1 Object Clipboard Tool



You can use the left mouse button to select any item in the Object Clipboard, then use the context menu (usually invoked by the right mouse button) to inspect, inspect class, open a Listener, or copy the object.

2

A Short Tutorial

This chapter gives you a short tutorial illustrating simple use of some of the major tools in the environment, and attempts to familiarize you with the way that tools can be used developing Common Lisp applications.

Note that some of the examples given in this chapter use symbols taken from the CAPI library. Do not worry if you are not familiar with the CAPI (if, for instance, you have been using another library, such as CLIM, to develop your applications). It is not essential that you fully understand the example code used in order to gain benefit from the tutorial. If you wish to learn more about the CAPI, you should refer to the *LispWorks CAPI User Guide* and the *LispWorks CAPI Reference Manual*, both of which are supplied in electronic form with your LispWorks software. The **Help** menu allows you to search all documentation from inside the Common LispWorks environment.

Note: When using the X11/Motif GUI, before you start working through the tutorial, ensure that the **DISPLAY** UNIX environment variable is set correctly, and that you have started Common LispWorks, for example by

```
(env:start-environment)
```

To maintain continuity, try to work your way through the whole of this tutorial in one session.

2.1 Starting the environment

On Linux and FreeBSD, assuming that you have the LispWorks executable location in your path, just type its name in any xterm or command shell window. This name is **lispworks-5-1-0-x86-linux**, **lispworks-5-1-0-amd64-linux** or **lispworks-5-1-0-x86-freebsd** depending on which product you are running. Under KDE or Gnome, you might want to set up a system menu item to start LispWorks.

On Solaris and HP-UX, the supplied image is in the **lib/5-1-0-0/config** directory of the LispWorks installation. You may have a configured image elsewhere.

On Linux and FreeBSD, the LispWorks environment starts automatically in the supplied image. On Solaris and HP-UX, the LispWorks environment starts when the command line argument **-env** is specified. If **-env** is not specified, LispWorks will start in terminal ("tty") mode with a prompt similar to the following:

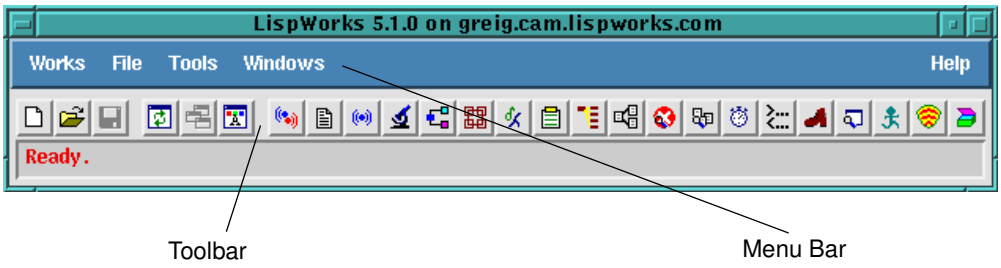
CL-USER 1 >

Type the following Lisp form at the prompt to start the LispWorks environment:

```
(env:start-environment)
```

After a short pause, you should see a splash screen, followed by the Podium window. The Podium is shown in Figure 2.1. A Listener window will also appear if your image is configured to start one.

Figure 2.1 The Podium



The Podium window is automatically displayed whenever you start the Common LispWorks environment. Its menu bar gives you access to various commands, as well as all the other tools in the environment. Its toolbar gives you quick access to some of the more convenient menu commands.

Like many other applications, the menu bar contains **File**,**Tools**, **Windows** and **Help** menus and a LispWorks specific menu named **Works**. The **Works** menu contains commands that apply to the current window and also contains menus that allow navigation between tools in the LispWorks environment.

The **File** menu allows you to open a file in an Editor, or print a file, regardless of which window is active. When the Editor or Listener tool is active, the **File** menu contains other commands for miscellaneous operations on the file displayed. The **Tools** menu gives you access to all of the Common LispWorks tools. The **Windows** menu lists all the active LispWorks windows you have running.

Note: If you wish to exit the Lisp image during this tutorial or at any other time, choose **Works > Exit > LispWorks**.

2.1.1 The Lisp Monitor

A Lisp Monitor window also appears when you start Common LispWorks. This is actually a separate process which shows you the state of the Lisp image, and monitors any garbage collection activity which occurs. For the most part you can ignore this window, although you may sometimes find the buttons on it useful for breaking into the Lisp process if you run source code which crashes Lisp for any reason. If you wish, you may close the Lisp Monitor window.

Figure 2.2 The Lisp Monitor

2.2 Creating a Listener

The Listener tool interactively evaluates the Lisp forms you enter. During a typical session, you evaluate pieces of code in the Listener, then examine the effects in other

tools, returning to the Listener whenever you want to evaluate another piece of. The structure of this tutorial reflects this two-stage approach.

Except on Unix platforms, a Listener is created when you start the Common Lisp-Works environment. If you don't currently have a Listener (check the **Windows** menu), start one by choosing **Tools > Listener** from the podium or clicking on in the Podium. This section of the tutorial demonstrates some of its more useful features. A Listener window is shown in Figure 2.3 below.

Figure 2.3 Listener

The Listener contains two views: the Listener view and the output view. At the bottom of the Listener is an echo area that is visible in either view. The echo area is used to prompt you for information when performing editor commands such as searching for text. You can switch between the two views by clicking the Listener and Output tabs respectively. You can evaluate Lisp forms in the Listener view by typing the form, followed by Return. Any output that is produced is displayed in the Listener view.

1. Type the following form into the Listener and press **`Return`**.

   ```
   (+ 1 2)
   ```

   The result of the evaluation, 3, appears in the Listener, and a new prompt is printed. Notice that the number in the prompt has been incremented, indicating that a form has been evaluated.

   Because you may want to enter a number of very similar forms, commands are provided which make this easy.

2. Press **Meta+P**.

   The form that you just evaluated is printed at the new prompt. You can press **Return** to evaluate this form again, or, more usefully, you can edit the form slightly before evaluating it.

3. Press **Ctrl+B** to move the cursor back one space. Now press the **Backspace** key to delete the number **2**, and type **3** in its place.

   You have edited the form **(+ 1 2)** to create a new form, **(+ 1 3)**.

4. Press **Return** to evaluate the new form.

The result of the evaluation, 4, appears in the Listener, followed by another new prompt, with the prompt number incremented once again.

2.3 Using the Debugger

A debugger tool is provided to help track down the cause of problems in your source code. This section introduces you to some of the ways in which it can be used.

1. Type the following definition into the Listener:

```
(defun test ()
  (let ((total 0))
    (loop for i below 100 do
      (incf total i) when (= i 50) do
      (break "We've reached fifty"))))
```

   This function counts from 0 to 100, accumulating the total as it progresses, and forces entry into the debugger when the count has reached 50.

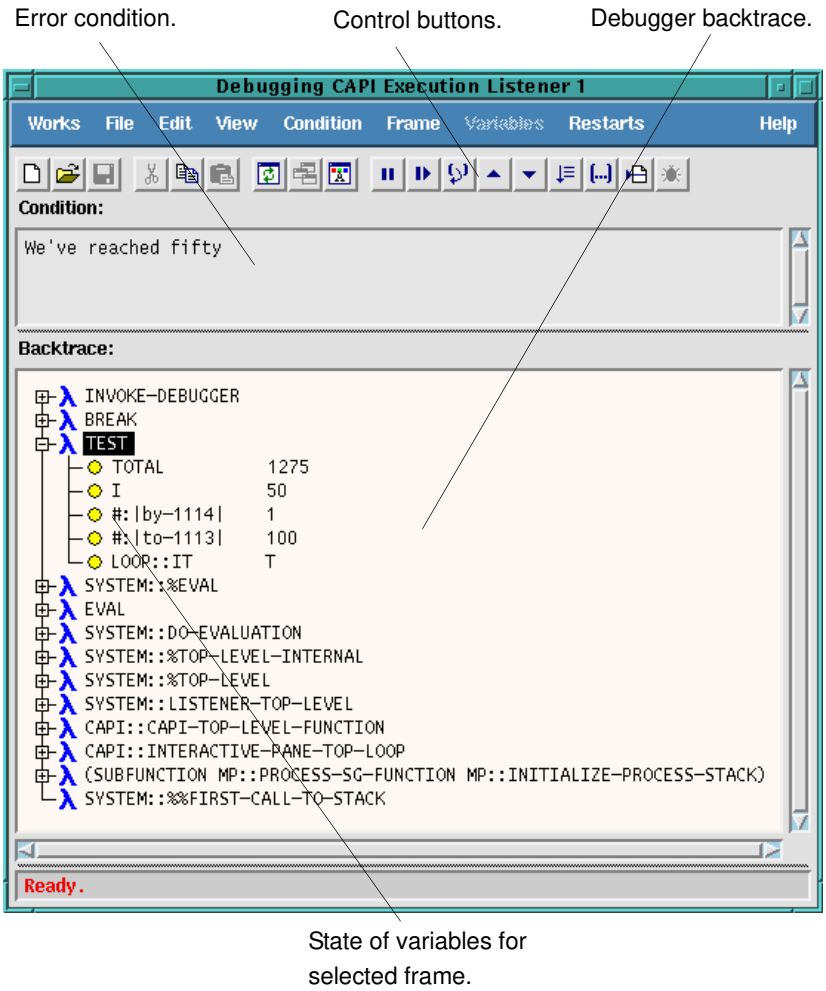
2. Next, call the function by entering **(test)** into the Listener.

   Initially, the command line debugger is entered. This is a debugger which can be used from within the Listener itself. More details about what you can do in the command line debugger can be found by typing **:?** at the debugger prompt.

3. To enter the debugger tool at this point, choose the menu command **Debug > Start GUI Debugger** or press in the Listener toolbar.

   The debugger tool appears, as shown in Figure 2.4.

Figure 2.4 Debugger tool



The debugger tool gives a view of the backtrace (in the Backtrace pane), showing the functions that are on the stack, and their internal variables (including any arguments) at the point that the error occurred.

4. In the Backtrace pane, notice that there is a rectangle with cross to the left of the word **TEST**. This indicates an expandable node. Click on this to open up the tree display, showing the local variables used in function **test**. Notice that the value for **i** is 50, as you would expect.

   There is a row of toolbar buttons at the top of the debugger which let you perform a number of different actions.

5. Choose **Restarts > (continue) Return from break.** or click on the Continue icon from the toolbar to exit the Debugger and continue execution.

The debugger disappears from the screen, and the command line debugger in the Listener is exited, leaving you at the Lisp prompt in the Listener.

2.4 Viewing output

There are many different ways to view output generated by the environment. In many tools, for example, output appears as soon as it is generated — this happens, for instance, when you compile code in the built-in editor.

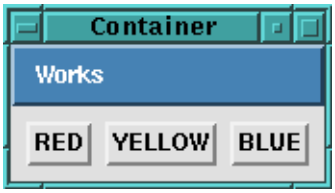
At other times, you can view output in a tool called the Output Browser. This tool collects together *all* the output generated by the environment, and is particularly useful for viewing output generated by your own processes (which cannot be displayed in any other environment tool). The Output Browser displays all the output sent to the default value of the variable ***standard-output***.

1. Evaluate the following in the Listener.

```
(capi:contain
 (make-instance 'capi:push-button-panel
                :items '(:red :yellow :blue)
                :selection-callback
                #'(lambda (data interface)
                    (format t
                            "Pressed button in interface ~S~%
data=~S~%"
                            interface data))))
```

   This is a piece of CAPI code that creates a window with three buttons, labeled **RED**, **YELLOW** and **BLUE**, as shown in Figure 2.5. Pressing any of these buttons prints the value of the button pressed.

Figure 2.5 Example CAPI window



2. Click on the Output tab in the Listener.
3. Try clicking on any of the buttons in the window you just created, and look at the output generated.
4. Now try a second example by entering the form below into the Listener at the current prompt (remember to click the Listener tab in the Listener first).

```
(capi:contain (make-instance
               'capi:text-input-pane
               :callback #'(lambda (text interface)
                             (format t
                                     "You entered: ~S~%" text))
               :title "My Text Input Pane"))
```

   The object that this code creates is going to demonstrate the Inspector tool. The code above creates a window containing a text input pane. You can type text directly into a text input pane, and this can be passed, for instance, to other functions for further processing.
5. Type the word **`hello`** into the text input pane and press **`Return`**. Look at the generated output in the output view.

2.5 Inspecting objects using the Inspector

The variables **`*`**, **`**`**, and **`***`** hold the results of expressions which have been evaluated in the Listener. **`*`** always holds the result of the last expression evaluated; **`**`** holds the previous value of **`*`**, and **`***`** holds the previous value of **`**`**. These variables (**`*`** in particular) are not only useful in their own right; the environment uses them to pass values between different tools.

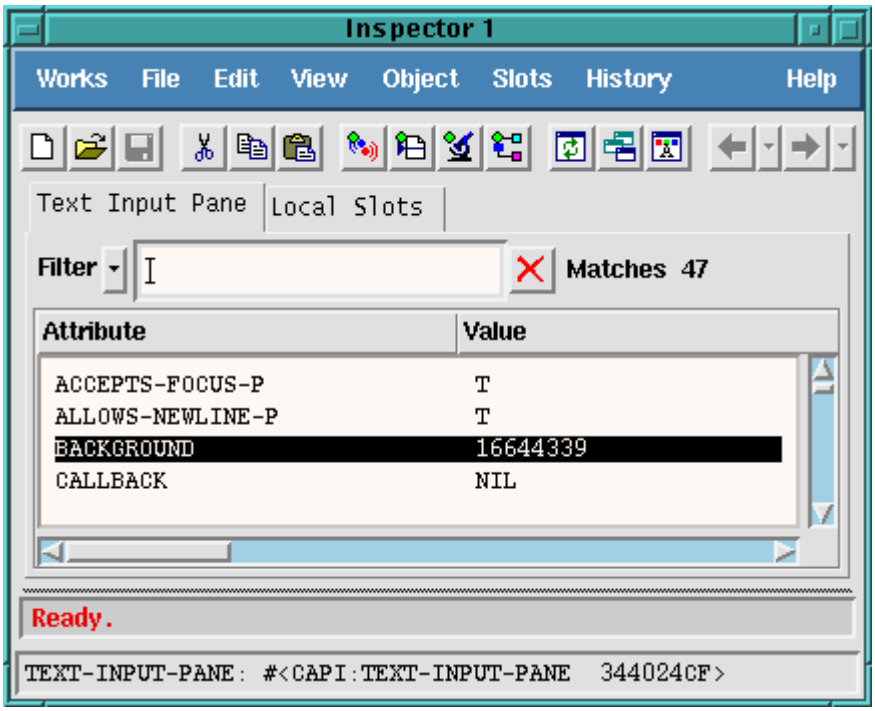
1. Make sure the Listener tab is visible, and type **`*`**.

If you have followed this tutorial so far, the text input pane object that you created above is returned. This is because the **`capi:contain`** function returns the object that is being contained. You can easily inspect this object more closely in the Inspector tool.

2. Choose the menu command **Values > Inspect**.

   This creates an Inspector tool which displays the **`capi:text-input-pane`** object currently contained in **`*`**.

Figure 2.6 Examining a text input pane in the Inspector



The commands in the **Values** menu always act upon the current value of **`*`**. This enables you to pass a value easily from one tool to another.

The main part of the Inspector is a list of all the slots in the object being inspected. This list shows both the name of each slot and its current value. Above this list is a box labeled Filter. This lets you filter the information shown in the main list, which can be useful when you are inspecting objects with a large number of slots. The name of the object being inspected appears immediately below the echo area.

3. Click in the Filter box, type the word **`text`**.

   This restricts the display in the Inspector to only those items which contain the string “text”, either in the slot name or in the slot value.

   After using the filter, you can easily see that one of the available slots contains the word **`hello`** that you typed into the text input pane.

   The Inspector always displays the actual instantiation of a given object (as opposed to a copy of it), so that you can be certain that any changes to the object itself are reflected in the Inspector.

4. Display the text input pane that you created earlier.

   If you can no longer see it, choose **Works > Windows > Container**; this is a simple way to display any of the windows and tools that you have created so far. (There are actually two windows with this name; if you choose the wrong one first of all, then just choose the other one.)

5. Click in the text input pane and delete the word **`hello`**. Type **`goodbye`** and press **`Return`**.

6. Select the Inspector to make it the active window and choose **Works > Refresh**.

   The description of the text slot now reflects the new value you specified.

7. Close the Inspector by choosing **Works > Exit > Window**.

You can close any window in the environment in this way, although there are often other ways of closing windows.

2.6 Examining classes in the Class Browser

This section shows you how to use the Class Browser tool to examine information about the Common Lisp class of any given object. The examples given use the text input pane object that you created earlier, and show you how you can change the values of a slot programmatically.

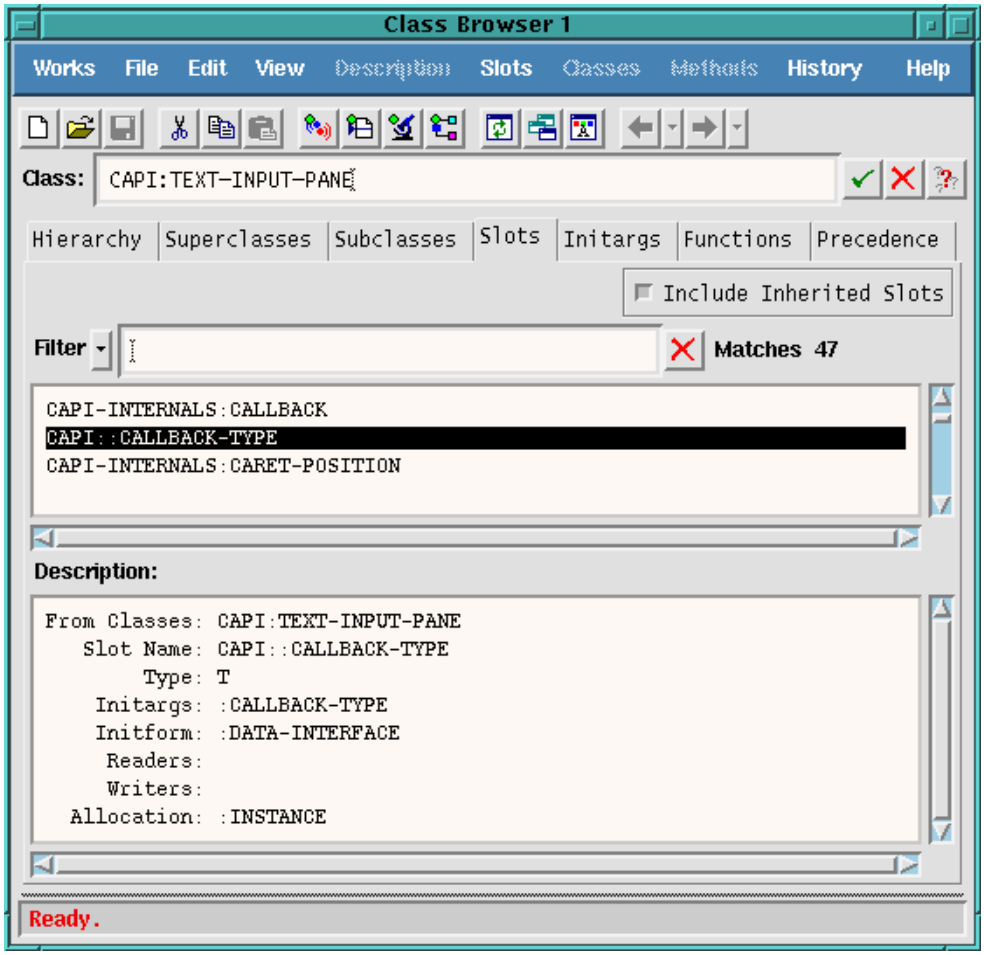
1. In the Listener, type **`*`** once again.

Notice that the * variable still contains the value of the text input pane object. This means that it is easy to perform several actions on that object. Notice further that the environment is aware that the object has been changed: the value returned by * reflects the change to the text slot that you made in the last section.

2. From the Listener, choose **Values > Class**.

This creates a Class Browser, shown in Figure 2.7, which allows you to examine the class of the object contained in *.

Figure 2.7 Examining the class of an object using the Class Browser



Ensure that the Slots tab is selected, as in the illustration. In the Class box, the name of the current Common Lisp class is printed. The list below the Filter box displays the slots available to the current class, and list labeled Description displays the description of any selected slot. The filter works in the same way as the Inspector's filter. There is also a checkbox labeled Include Inherited Slots. Clicking this checkbox lets you switch between displaying all the slots defined on the current class and all its superclasses, and only those slots defined directly on the current class. By default, slots defined on any superclasses (inherited slots) are shown in the main area.

3. Filter the display as you did for the Inspector; click in the Filter box, and this time type the word **foreground**.

   Only those slots with the string "foreground" in their names are displayed.

4. Select the **CAPI::FOREGROUND** slot from the list. A description of the slot appears in the description area, including information such as the initargs, readers, and writers of the slot.

Notice that the class text input pane has both a reader, **capi:simple-pane-foreground**, and a writer, **(setf capi:simple-pane-foreground)**. We can use this information to programmatically change the text shown in the text input pane.

5. Type this form into the Listener:

```
(setf (capi:simple-pane-foreground *) :red)
```

The text displayed in the text input pane is displayed in red to reflect the new value you have specified. Notice how you were able to use the ***** variable to refer directly to the text input pane object itself.

2.7 Summary

In this introductory tutorial you have seen how to perform the following actions:

- Start the windowing environment.
- Evaluate and re-evaluate Common Lisp forms using the Listener.
- Invoke the Debugger, follow the backtrace that it produces, and return from the error which caused entry to the Debugger.
- Collect and display data generated by your own code in the Output Browser.
- Use the Inspector to examine the current state of an object.
- Use the Class Browser to find out detailed information about a given class, so that you can make arbitrary programmatic changes to an instance of that class.

The next two chapters describe elements of the environment which are common to all tools.

Other chapters in this manual describe the other tools available in the environment. Each chapter is intended to be reasonably independent of the others, so you can look at them in any order you wish. You are advised to study the chapters on the basic tools,

such as the Inspector, the Class Browser and the Editor first, since a knowledge of these tools is vital if you want to get the best out of the environment.

3

Common Features

The Common LispWorks environment has been designed so that its features are consistent throughout, and tools have a uniform look and feel. All tools have certain characteristics which look the same, and behave in a consistent manner. By making as many common features as possible, learning how to use each tool is much simpler.

Chapter 2, “A Short Tutorial”, introduced you to some of the major tools in the environment, demonstrating the commonality and high integration between them, and showing how this can be used to good effect in the development process. This chapter describes these common features in more detail.

Most of the common features in the environment can be found under the **Works**, **File**, **Tools**, **Windows**, **History** and **Help** menus. Using the commands available under these menus you can:

- Move to any other tool.
- Cut, copy or paste via the clipboard and the Object Clipboard tool.
- Perform search and replace operations.
- Re-issue a previous command, or re-examine an object.
- Perform operations such as loading and saving files.

Each menu command operates on the window associated with the menu.

In addition, some other conventions have been adopted throughout the Common LispWorks environment:

- Many tools have a number of different views: ways of displaying information. Each view is made available by clicking on a different tab in the tool.
- Lists displayed in many tools can be filtered in order to hide redundant or uninteresting information.

These features are described in full in this chapter. Please note that subsequent descriptions of individual tools in the environment do *not* include a description of these menus, unless a feature specific to the individual tool is described.

Online help is also available from the **Help** menu in any window. These facilities are described in Chapter 4, “Getting Help”.

Many tools allow you to display information in the form of a graph. These graph views behave consistently throughout the environment, and a description of the graph features offered is given in Chapter 5, “Manipulating Graphs”.

3.1 Displaying tool windows

There are many tools available, and you can display them in a number of ways.

You can also control how tools are re-used within the environment. That is, whether an existing Listener window (for example) is raised or a new one created, when you ask for a Listener tool. In this section we will discuss global and per-tool control of reuse.

3.1.1 Displaying existing windows

Choose the **Windows** menufrom the podium. This menu contains a list of all the windows currently available in the environment. Choosing any item from this list brings the window to the front of the display.

3.1.2 Iconifying existing windows

To iconify a window, use the command provided by your window manager.

3.1.3 Displaying tools using the mouse

To display most tools:

1. Choose the **Tools** menu from the podium.

   Most tools in the environment are listed in this menu.

2. Choose the tool you require from the menu.

or

1. Choose the **Works > Tools** menu from any tool.
2. Choose the tool you require from this menu.

or

1. Click the appropriate button on the Podium.

   For example, to display a Process Browser, click .

The tool is created (if necessary), and displayed. Using this method can be useful you may not remember immediately whether you have an existing instance of a given tool or not.

3.1.4 Displaying tools using the keyboard

Accelerators are provided for the popular items on the **Tools** menu.

Each tool accelerator is an alphanumeric key together with platform-specific modifier keys as shown in “Tool accelerator keys” on page 22.

You cannot configure these pre-defined *tool accelerators*.

3.1.4.1 Tool accelerator modifier keys

On Motif the Modifiers are **`Meta+Ctrl`**. For example, **`Meta+Ctrl+L`** raises a Listener.

3.1.4.2 Tool accelerator keys

The accelerator keys for each tool are as shown in Table 3.1

Table 3.1 Tool accelerators

Tool Name	Accelerator
Listener	**Meta+Ctrl+L**
Editor	**Meta+Ctrl+E**
Output Browser	None
Inspector	**Meta+Ctrl+I**
Class Browser	**Meta+Ctrl+C**
Generic Function Browser	**Meta+Ctrl+G**
Symbol Browser	**Meta+Ctrl+S**
Object Clipboard	**Meta+Ctrl+O**
Function Call Browser	**Meta+Ctrl+X**
System Browser	**Meta+Ctrl+Y**
Compilation Conditions Browser	**Meta+Ctrl+D**
Search Files	**Meta+Ctrl+F**
Profiler	None
Tracer	**Meta+Ctrl+T**
Stepper	None
Window Browser	**Meta+Ctrl+W**
Process Browser	**Meta+Ctrl+P**
Shell	None
Application Builder	**Meta+Ctrl+A**

3.1.4.3 Special considerations when using tool accelerators

On Motif tool accelerators work only in KDE/Gnome editor emulation.

Also, you need a keyboard with **`Alt`** on **`mod1`** and **`Meta`** on a different modifier (for example, **`mod3`**).

3.1.5 Re-using tool windows

3.1.5.1 Global control of re-use

By default, tools windows are re-used where possible. For example, suppose you already have a Listener window (potentially iconified) but do not have an Inspector window. When you choose **Tools > Listener** in the podium, the existing Listener is displayed. When you choose **Tools > Inspector**, an Inspector is created and displayed.

You can switch off re-use of tool windows via the command **Works > Tools > Global Preferences...**. Uncheck the **Reuse all tools** box and click **OK**. Now, when you choose **Tools > Listener** a new Listener is created, regardless of whether one already exists, and other tools behave in the same way.

The setting of **Reuse all tools** will be retained for your subsequent LispWorks sessions.

3.1.5.2 Per-window control of re-use

When the **Reuse all tools** option is on, tools windows are reusable by default. However, it is possible to specify that a particular instance of a tool is not reusable. To make your Inspector not reusable, follow these steps:

1. Ensure that the **Reuse all tools** option is checked under **Works > Tools > Global Preferences....**
2. In your Inspector window, open the menu **Works > Customize** and deselect the **Resuable** option.
3. Now try **Tools > Inspector**. A new Inspector window is created.

The **Reuse all tools** option is persistant, but the per-tool setting **Resuable** applies only to the current instance of the tool, and it does not affect future sessions.

3.1.6 Toolbar configurations

Most tools have toolbars offering one-click access to frequently-used commands. For example, the Editor has a toolbar for operating on source code.

Figure 3.1 The Editor's source operations toolbar

You may prefer to remove such toolbars. You can control whether a tool displays its toolbars by the option **Show Toolbar**.

To hide toolbars for the active tool:

1. Choose **Works > Tools > Preferences...**.

   The Preferences dialog for the active window appears.

2. Select the **General** tab.

3. Uncheck **Show Toolbar** and press **OK** to confirm the dialog.

Note: The functionality of each toolbar is available elsewhere. For example the Editor's source code operations are also available on the **Buffer**, **Definitions** and **Expression** menus.

3.1.7 Copying windows

Choose **Works > Clone** in a given tool window to make a copy of that tool window. This is useful, for instance, if you wish to have two different views on an object simultaneously, and allows you to have several copies of a tool without having to change its re-use property using the **Works > Customize** menu.

3.1.8 Closing windows

Close any window in the environment using one of the following methods:

- Choose **Works > Exit > Window**
- Use a window-manager-specific feature, if available
- In Editor windows only, use the Emacs-like command **`Delete Window`** (keystroke **`Ctrl+X 0`**)

3.1.9 Updating windows

To manually update any tool, choose **Works > Refresh**.

Updating a tool is a useful way of making a snapshot of an aspect of the environment that you are interested in. For instance, imagine you want to compare a number of instances of a CLOS class against a known instance of the same class using the Inspector. You can do this as follows:

1. Create an object to inspect, by entering in a Listener

   ```
   (make-instance 'capi:text-input-pane)
   ```

2. Choose **Values > Inspect** to view the object in the Inspector.
3. Make sure the Inspector is the active window, and choose **Works > Clone** to make a copy of it.
4. In the Listener, enter the same form again to create a second object.

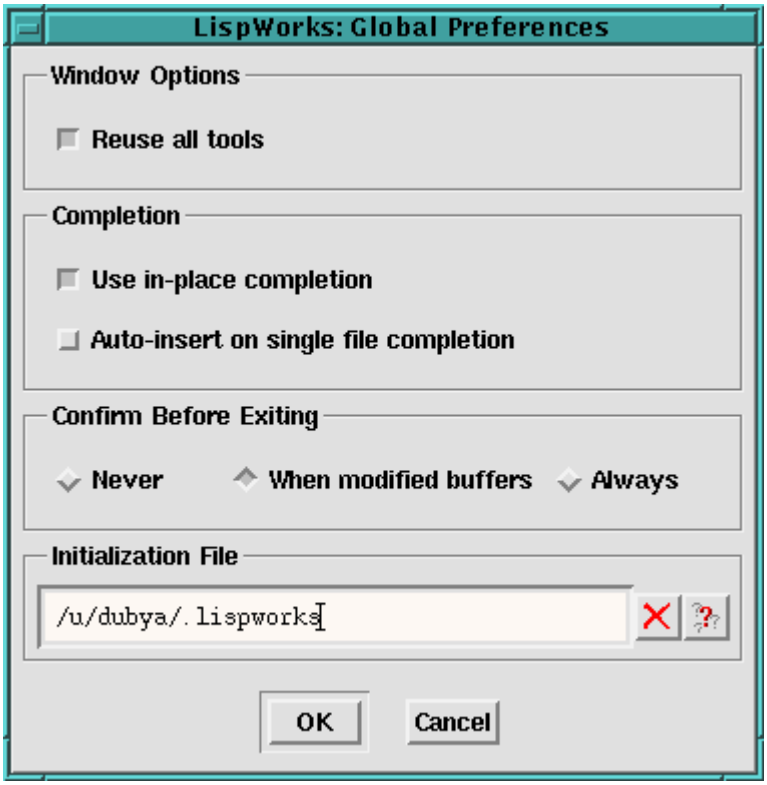
   Note: You can use **`Esc P`** in Emacs emulation or **`Ctrl+Up`** in Windows emulation to get the previous Listener command.
5. View the new object in the Inspector as in Step 2. Compare it to the original instance that is still displayed in the clone.

3.2 Setting global preferences

Choose **Tools > Global Preferences...** from the podium or **Works > Tools > Global Preferences...**to invoke the global preferences dialog. This dialog is used to specify global preferences, such as the name of the initialization file to use (which is initially **`.lispworks`**).

Note: it is up to each user to create and maintain their own personal initialization file. A sample personal initialization file is supplied with LispWorks - see the file **`lib/5-1-0-0/config/a-dot-lispworks.lisp`** in the LispWorks distribution.

Figure 3.2 The Global Preferences dialog



3.2.1 The window options

Checking **Reuse all tools** ensures that LispWorks uses an existing tool rather than starting up a new copy. For example, if an editor is already open, choosing **File > Open** and selecting a new file causes the file to be opened in the existing editor.

3.2.2 Controlling completion behavior

In-place completion is enabled by default in the IDE. If you prefer the modal dialog style of completion familiar to users of LispWorks 5.0 and previous versions, deselect the **Use in-place completion** option.

When using in-place completion to complete a filename, by default you must always select an item from the in-place completion window. You can accelerate this interaction by checking the option **Auto-insert on single file completion**. Then, if there is just one possible completion, it is automatically selected and appended to your input.

3.2.3 Quitting the environment

Choose **Works > Exit > LispWorks...** to exit LispWorks.

You can control whether LispWorks prompts for confirmation before exiting, using **Works > Tools > Global Preferences...**. The **Confirm Before Exiting** preference has these meanings:

Never LispWorks exits immediately.

When modified buffers
If there are modified editor buffers, a dialog asks you whether these should be saved before exiting.

Always A dialog asks you to confirm whether LispWorks should exit.

3.2.4 Initialization file

By default LispWorks looks for a file `.lispworks` to be loaded automatically when LispWorks is started. You should create an initialization file and add to it Lisp code to initialize the LispWorks image to suit your needs.

The global preferences dialog can be used to specify a different initialization file. You can either enter the path and filename directly into the text input box, or use the button to display a file selection dialog. Clicking on undoes any alterations entered.

3.3 Performing editing functions

This section discusses commands available in the **Edit** menu of any window. These commands fall into five areas:

- Undoing changes.
- Using the clipboard.
- Selecting text and objects.
- Searching for text.
- Substituting text.

3.3.1 Undoing changes

You can undo changes made in a tool using **Edit > Undo**. This facility is most useful in the Editor and Listener— see "Other essential commands" on page 169 for more details.

3.3.2 Using the clipboard

You can use the clipboard to transfer data between tools, or even between the environment and other applications that you are running. There are three commands available, as follows:

- Choose **Edit > Copy** to put the selection or "primary object" onto the clipboard.
- Choose **Edit > Cut** to put the selection or "primary object" onto the clipboard and remove it from the tool it was copied from.
- Choose **Edit > Paste** to put the contents of the clipboard into the current tool.

Unlike the clipboard in many other applications, the Common LispWorks clipboard can contain a Common Lisp object. This makes the Common LispWorks clipboard an exceptionally powerful tool, allowing you to pass objects between different tools in the environment so that they can be examined in different ways.

Use of **Copy** or **Cut** followed by **Paste** lets you transfer items between tools, or to different parts of the same tool. There are several ways to use these commands:

- In the Class Browser (for example) you can **Copy** the class to the clipboard and then **Paste** it into another tool. Because the Common Lisp object itself is cop-

ied to the clipboard, it is treated usefully according to the tool. For instance, if you paste it into an Inspector, it is inspected. If you paste it into an editor however, the class name is simply pasted as text.

- In the Editor, you can **Copy** chunks of text and **Paste** them into different places, either within the same file or between different files. If you have sections of code which are very similar, rather than typing each section out explicitly, just **Paste** in the same section as many times as you need and change only the relevant parts. Section 12.11 on page 171 describes a number of more sophisticated methods that can be used in the Editor.
- Between any of the tools, you can **Cut**, **Copy**, and **Paste** Common Lisp objects. You can, for instance, make an instance of a class in the Listener, inspect it by **Values > Inspect**, and then **Copy** it in the Inspector, and then **Paste** it into a Class Browser to examine its class.
- If you have several Common Lisp objects which you want to keep track of, store them in the Object Clipboard. You can do this by a **Clip** command in tools such as the Class Browser, or by **Edit > Paste** in the Object Clipboard tool. See Chapter 8, "The Object Clipboard" for more information about that tool.

As well as the menu commands, you can use the , and buttons in the toolbar, for **Cut**, **Copy** and **Paste** respectively.

Note: You can also transfer data within the environment using the standard actions commands described in Section 3.7 on page 35.

UNIX Implementation Note: The environment also interacts with the standard UNIX clipboard, so that data can be transferred to or from applications other than Lisp. To do this, the UNIX and Common LispWorks clipboards are kept in synchronization all the time, as follows:

- Whenever a Common Lisp object is copied to the Common LispWorks clipboard, its string representation is copied onto the UNIX clipboard.
- Whenever a string is copied to the UNIX clipboard, it is copied onto the Common LispWorks clipboard as a string.

3.3.3 Selecting text and objects

Choose **Edit > Select All** or **Edit > Deselect All** to select or deselect all the text in an Editor or Listener window, or all the items in a list or graph. These commands are useful whenever there is too much information to be able to select items one at a time.

3.3.4 Searching for text and objects

You can search for and change text in most tools using **Edit > Find...**, **Edit > Find Next**, and **Edit > Replace...**.

Choose **Edit > Find...** to find an item in the current tool (this might be a piece of text, or a fragment of Common Lisp, or an object, depending on the tool). You must supply an item to find in the dialog that appears.

Choose **Edit > Find Next** if you want to search for the next occurrence of an item you have already found. This command does not prompt you for an item to find, and so is only available if you have already found something.

Choose **Edit > Replace...** if you want to replace one string of text with another. A dialog box prompts you for a text string to find, and a text string to replace it with. This command is only available in the Editor and the Listener, and is most useful in the Editor.

3.4 The history list

The *history list* of a tool stores the most recent events which have been carried out in that tool, or the most recent objects which have been browsed in it.

The **History > Items** submenu provides a list of these events (or objects), allowing you to repeat any of them (or browse them again) by choosing them from the menu. This gives you an easy way of repeating forms in the Listener, inspecting objects or browsing classes again, revisiting searches, and so on.

The menu lists the last ten *unique* items to have entered the history list of the active window. Because each entry is unique, some items may have occurred more than ten events ago.

If the editor is the active window, the **History > Items** submenu lists the buffers currently open.

3.4.1 Repeating events from the history list

The easiest way of repeating an event from the history list is to choose it from the **History > Items** submenu. There may be times, though, when this is inconvenient (the items on the list may be too long to be able to distinguish between them easily, or you might want to repeat an item that occurred more than ten events ago). In such cases, there are three commands which offer an alternative way of choosing items.

Choose **History > Previous** to perform the previous item in the history list of the tool. This is usually the most recent event you have performed, but may not be (if, for instance, the last action was itself an event that was already on the history list).

Choose **History > Next** to perform the next item in the history list. This item is not usually available unless the last event you performed involved an item already on the history list.

Note: You can also use the and buttons in the toolbar.

3.4.2 Editing the history list

Choose **History > Modify** to remove items from the **History > Items** menu. A dialog appears that contains all of the items in the current **History** menu. Select the items you wish to *retain*, and click **OK**. Any items which were *not* selected in the dialog are removed from the history list.

Note: another way to keep track of items that you're interested in (such as appear in the history lists of various tools) is to place them on the Object Clipboard. See Chapter 8, "The Object Clipboard" for more details.

3.5 Operating on files

The **File** menu allows you to perform operations on files stored on disk. Some commands are only available for tools which need to interact with the files you have stored on disk, such as the Listener and Editor.

The default commands available in the **File** menu are described below. Note that in some tools, the **File** menu contains additional commands specific to that tool. Please refer to the relevant chapters for each tool for a description of these additional commands.

Choose **File > New** to open a new buffer in the built-in Editor. If an Editor window has not yet been created, this command also creates one. The new buffer is unnamed.Alternatively, you can click the button in the toolbar. This toolbar button is available on appropriate tools, and in the podium as shown in Figure 2.1, page 6.

Choose **File > Open** to open an existing file in a new editor buffer. Where appropriate, a dialog appears, allowing you to choose a filename. If an editor window has not yet been created, this command creates one. Alternatively, you can click the button in the toolbar. This toolbar button is available on appropriate tools, and in the LispWorks podium, shown in Figure 2.1, page 6.

Choose **File > Load** to load a file of Lisp source code or a fasl (binary) file. Choose **File > Compile** to compile a file of Lisp source code. Choose **File > Compile and Load** compile a source file *and* load the resulting fasl file. When appropriate, each command displays a dialog, allowing you to choose the file you want to load or compile.

Choose **File > Print** to print a file. A dialog allows you to choose a file to print. The current printer can be changed or configured by choosing the **File > Printer Setup...** menu option.

Choose **File > Browse Parent System** to view the parent system of the current file in the System Browser. This command is only available if the system has already been defined. See Chapter 26, “The System Browser” for a complete description of the System Browser.

Choose **File > Recent Files** to raise a submenu listing the last 10 files visited via the **File > Open...** and **File > Save As...** commands. This allows speedy return to the files you are working on.

Note: As described above, the behavior of each command can vary slightly according to the tool in which the command is chosen. For instance, choosing **File > Print** in the Editor prints out the displayed file, whereas choosing **File > Print** in the Listener prompts you for a file to print.

3.6 Displaying packages

Symbols can be displayed either with their package information attached or not. In the LispWorks IDE, symbols are displayed with the package name attached by default.

For example, suppose you have created a package **FOO** which includes a symbol named **bar** and a symbol named **baz**. Suppose further that you created a new package **FOO2**, which used the **FOO** package. This can be done as shown below:

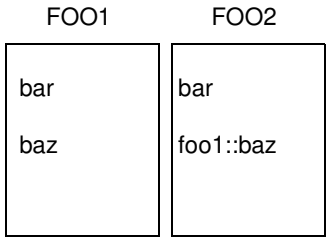
```
(defpackage foo (:use "COMMON-LISP"))
(defpackage foo2 (:use "FOO" "COMMON-LISP"))
```

Note that in defining both packages, the **COMMON-LISP** package has also been used. It is good practice to use this package, to ensure that commonly-used symbols are available.

When creating packages which use other packages, exported symbols can be called without having to refer to the package name.

To illustrate this, let us return to our example.

Figure 3.3 Two example packages



We have two packages: **FOO1** and **FOO2**. **FOO1** contains symbols **bar** and **baz**. The symbol **bar** has been exported, whereas the symbol **baz** is not exported.

When the current package is **FOO2**, you can refer to **bar** without using the package name. This is because **FOO2** uses **FOO1** and **bar** is exported. However to refer to **baz** you must still use the **FOO1** package name like this: **foo1::baz**. This is because **baz** is not exported.

Note also that when the current package is other than **FOO1** or **FOO2**, you can refer to **foo1:bar**, but you can only refer to **baz** as **foo1::baz**.

Package names are usually displayed alongside symbols in a list. Having a package entry on every line can be unhelpful, especially if the majority of items listed are from the same package. To hide the package names for the symbols in the active window:

1. Choose **Works > Tools > Preferences...**.

   The Preferences dialog for the active window appears.

2. Uncheck **Show Package Names** in this dialog.

3.6.1 Specifying a package

If you are working in a particular package, you can adjust the current tool to display its symbols as you would refer to them from that package—that is, as the package sees them. This can make listings clearer and, more importantly, can show you which symbols have been exported from a package.

Doing this changes the process package of the tool. This means that both displayed symbols and symbols typed into the tool are assumed to be in the package specified. This can be useful in a browser, for example, if you intend to browse a number of different objects which come from the same package.

To change the process package for the active window:

1. Choose **Works > Tools > Preferences...**.
2. Delete the package name in the Package box, and type in the name of the new package.
3. Click ✓ to confirm this new name.
4. Click **Apply** or **OK** to make the change.

Note: If you wish, you can partially type the package name and then click ?. This allows you to select from a list of all package names which begin with the partial input you have entered. See “Completion” on page 46 for detailed instructions on using completion.

As an example, imagine you are looking at a list of symbols in the Inspector. You are working in the package **FOO**, and some of the symbols in the Inspector are in that package, while others are in another package. To change the current package of the Inspector to **FOO**, follow the instructions below:

1. Choose **Works > Tools > Preferences...**.

   The Preferences dialog indicates that **COMMON-LISP-USER** is the current package in this window.
2. In the Package box, delete **COMMON-LISP-USER**, and type **FOO**.
3. Click **OK** to make the change.

In the Inspector all the symbols available from **FOO** appear without the package prefix **FOO**. Similarly, all exported symbols in packages which **FOO** uses appear without a package prefix, while all others have an appropriate package prefix.

3.7 Performing operations on selected objects

In any tool, there are a number of operations that you can always perform on the selected objects, irrespective of the type of objects you have selected. This allows you to perform some powerful operations and also ensures a consistent feel to every tool in the environment.

In this context the term "selected objects" is meant in the widest sense, and can refer to *any* items selected *anywhere* in a tool, be it in a list of items, or a graph. It can also refer to the tool's *current object*: that is, the object which is currently being examined.

These operations are available throughout the environment, and are referred to as standard action commands. As with other commands that are specific to the active window, standard action commands are usually available from menus on the main menu bar of the tool you are using. The objects which are operated on by a given standard action command depend on the menu from which you chose the command.

As an example, consider examining the contents of Common Lisp objects using the Inspector.

The standard action commands for the Inspector are present in two places: the **Object** menu, and the **Slots** menu.

- Choose a standard action command from the **Object** menu to perform an operation on the Inspector's current object.
- Choose a standard action command from the **Slots** menu to perform an operation on the selected components of the Common Lisp object.

Notice that in the first case, the object operated on is the tool's current object: you do not have to take any further action before performing the operation.

In the second case, the objects examined represent more specific pieces of information: you need to select them before you can perform the operation. This, therefore, examines more discrete pieces of information about the current object.

Many tools have one or more submenus like those described above. The first operates on the current object. What that object is, and hence the name of the submenu in

which the commands are to be found, depends on the tool you are using. For instance, if you are examining classes, the commands can be found in a **Classes** menu. If you are examining methods, they can be found in a **Methods** menu.

Some tools contain two or more such menus; precise details are given in the relevant chapters.

As a guide, if a menu has a plural for a name, the commands in that menu can be performed on multiple selections. If the menu name is not pluralized, commands only affect a single selection.

3.7.1 Operations available

The standard action commands available are described below. In these descriptions, the term “current object” refers to the Lisp object that is being acted upon by the menu command. This depends on the tool being used and the menu in which the command appears, but should be obvious from the context.

Choose **Browse** to browse the current object using an appropriate browser. A browser is a tool which lets you examine a particular type of Common Lisp object, and there are a large number of them available in the environment. Some of the browsers available are:

- The Class Browser, which lets you examine CLOS classes.
- The Generic Function browser, which lets you examine the generic functions in the environment, and the methods you have defined on them.

See the appropriate chapters for a full description of each browser; there is a chapter of this manual devoted to each available browser. The precise name of the **Browse** menu command reflects the type of browser that is used to examine the selected object. Thus, if the command is **Browse – Generic Function**, a Generic Function Browser is used.

Choose **Class** to look at the class of the current object in a Class Browser. Alternatively, click on in the toolbar. See Chapter 7, “The Class Browser” for full details about this tool.

Choose **Clip** to add the current object to the Object Clipboard. See Chapter 8, “The Object Clipboard” for full details about this tool.

Choose **Copy** to copy the current object to the clipboard, thus making it available for use elsewhere in the environment. Note that performing this operation on the object currently being examined by the tool (for example, choosing the command from the **Object** menu when an Inspector is the active window) has the same effect as choosing **Edit > Copy**, whereas choosing this option from other menus (such as a **Description** menu) copies more discrete information to the clipboard.

Choose **Documentation** to display the Common Lisp documentation (that is, the result of the function `documentation`) for the current object. It is printed in a help window.

Choose **Find Source** to search for the source code definition of the current object. Alternatively, click on in the toolbar. If it is found, the file is displayed in the Editor: the cursor is placed at the start of the definition. See Chapter 12, “The Editor” for an introduction to the Editor tool. You can find only the definitions of objects you have defined yourself (those for which you have written source code)—not those provided by the environment or the Lisp implementation.

Choose **Inspect** to invoke an Inspector on the current object. Alternatively, click on in the toolbar. See Chapter 16, “The Inspector”, for details about the Inspector. If you are ever in any doubt about which object is operated on by a standard action command, choose this command.

Choose **Listen** to paste the current object into the Listener. Alternatively, click on in the toolbar. Chapter 20, “The Listener” provides you with full details about this tool.

Choose **Function Calls** to describe the current object in a function call browser. See Chapter 13, “The Function Call Browser” for more details.

Choose **Generic Function** to describe the current object (a generic function or a method) in a Generic Function Browser. If the current object is a method, then its generic function is described in the Generic Function Browser and the method is selected. See Chapter 14, “The Generic Function Browser” for more details.

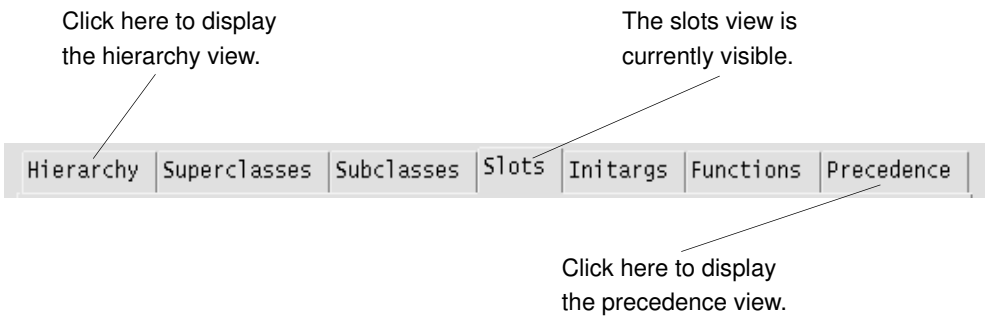
Choose **Browse Symbols Like** to display symbols matching the current object in a Symbol Browser. See Chapter 17, “The Symbol Browser” for more details.

3.8 Using different views

Every tool in the environment has several different views, each of which can display information which is pertinent to the task at hand. You can switch to any of the avail-

able views by clicking on the appropriate tab at the top of the tool. When choosing a different view, the layout of the tool itself changes.

Figure 3.4 Click tabs to display different views of a tool



In tools which are browsers, different views allow you to display different pieces of information about the same objects; for instance, in the Class Browser you can switch from a view which shows you information about the slots in a given Common Lisp class to one which shows information about the initargs of the class.

In other tools, different views may show you completely different types of related information. For example, in the Listener you can switch from the Listener view to a view that shows you any output that has been generated by the Listener.

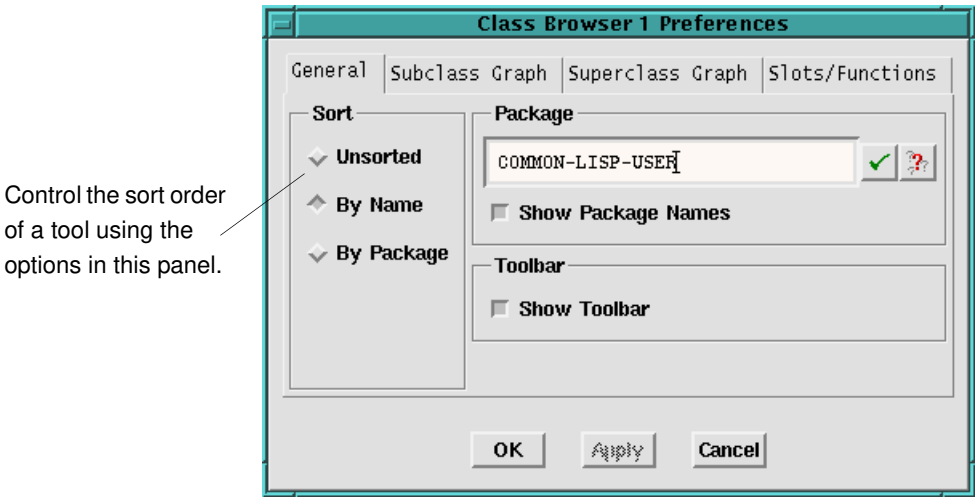
All tools have a default view when you first start them. The default view is the one which you are most likely to make most use of, or the one which you use first. When you first start the built-in Editor, the default view is the text view. When you start a Class Browser, the default view shows you the slots available for the current class, as you have already seen.

3.8.1 Sorting items in views

You can sort the items displayed in the main area of any view using the Preferences dialog for a given tool. In tools where items can be sorted in this way, there are at least the following three commands available:

1. For any tool, choose **Works > Tools > Preferences...** or click . Click on the General tab, if necessary.

Figure 3.5 Example Preferences dialog



Notice that some Preference dialogs, such as the one shown above, have several tabs. In these cases, the options described in this section are always available in the General tab.

The left hand side of the Preferences dialog is labeled Sort.

2. Choose one of the options in this area of the dialog to control the sort order of items in the active window.

   The options available vary according to the tool, but at least the following will be available:

By Name — Sorts symbols in a list or graph according to the name of each item. The packages that the symbols are resident in are ignored when this option is used; thus, the symbol **`vv:allocate`** would be listed before **`aa:vectorize`**.

By Package — Sorts symbols in a list or graph according to the package they are listed in. Thus, all symbols in the **`aa`** package would be listed together, as would all symbols in the **`vv`** package. In addition, the **`aa`** package would be listed before the **`vv`** pack-

age. Within a given package, objects are listed in alphabetical order when using this option: thus, **`aa:carry-out-conditions`** would be listed before **`aa:vectorize`**.

Unsorted Lists all symbols in a graph or list in the order in which they are defined in the source code. This can sometimes be a useful option in itself, and is always the quickest option available. You may sometimes want to use this option if you are displaying a large number of items and you are not filtering those items in any way.

Note: There are sometimes other sort options available in the Sort area of the Preferences dialog, depending on the nature of the tool itself. These options are described in the chapter specific to each tool.

Only those views whose main area consists of a list or a graph can be sorted. In particular, the default views of tools such as the Listener or the editor, which is an editor window which you can type directly into, cannot be sorted.

3.9 Tracing symbols from tools

For some tools, submenus under the relevant main menus (for example, the **Expression** menu on the Editor tool) contain a **Trace** submenu that allows you to set tracing options for a function, method, macro, or generic function. This is a useful shortcut to the **`trace`** macro, since it gives you some control over tracing in the environment without having to work directly at the Common Lisp prompt.

Below, the *current function* means the currently selected function, method, macro or generic function, or in the case of the Editor and Listener, the symbol under the cursor.

A **Trace** submenu generally has the following commands:

- Choose **Trace** to trace the current function.
- Choose **Trace Inside** to trace the current function within the current context. Choosing this command sets the **`:inside`** option for **`trace`**.
- Choose **Trace with Break** to trace the current function, and enter the debugger on entry to it. Choosing this command sets the **`:break`** option to **`t`**.
- Choose **Untrace** to turn off tracing on the current function.

- Choose **Untrace All** to turn off tracing on currently traced functions. Note that this does not turn off tracing in the environment as a whole.
- Choose **Show in Tracer** to trace the current function and display in the Tracer tool. This offers you more control over tracing. See Chapter 11, “The Tracer” for details.
- Choose **Toggle Tracing** to turn all tracing commands in the environment on or off. Choose **Toggle Tracing** again to restore the previous tracing state.

3.10 Linking tools together

You can link together pairs of tools, so that changing the information displayed in one tool automatically updates the other. This can be done for virtually any tool in Common LispWorks, and provides a simple way for you to browse information and see how the state of the Lisp environment changes as you run your code. For instance, you can make between an Inspector and a Listener so that every time you evaluate a form in the Listener, its value is automatically inspected.

You can also link two copies of the same tool. This can be a very useful way of seeing two views of a tool at once. For instance, you could create a copy of the Class Browser by choosing **Works > Clone**, and then link them together. By keeping one browser in the subclasses view, and the other in the slots view, you can automatically see both the subclasses *and* the available slots for a given class.

Linked tools have a master-slave relationship. One tool (the slave) gets updated automatically, and the other tool (the master) controls the linking process. To link together any two tools:

1. Select the tool that the link is to be established to. For example, to form a link from an Inspector to a Class Browser to ensure that a class selected in the Class Browser is automatically inspected, you would use the **Edit** menu of the Class Browser.
2. Choose **Edit > Link >** *fromtool* where *fromtool* is the title of the tool you wish to link from.

To break a link, select **-- No Link --** instead of a specific tool.

To view all the current links that have been established, choose **Edit > Link from > Browse Links...** Select any of the links listed and click on **Remove Link(s)** to remove them.

3.11 Filtering information

Many tools have views which display information in a list. Items in these lists may be selected, and you can usually perform operations on selected items (for instance, by means of the standard action commands, as described in “Performing operations on selected objects” on page 35).

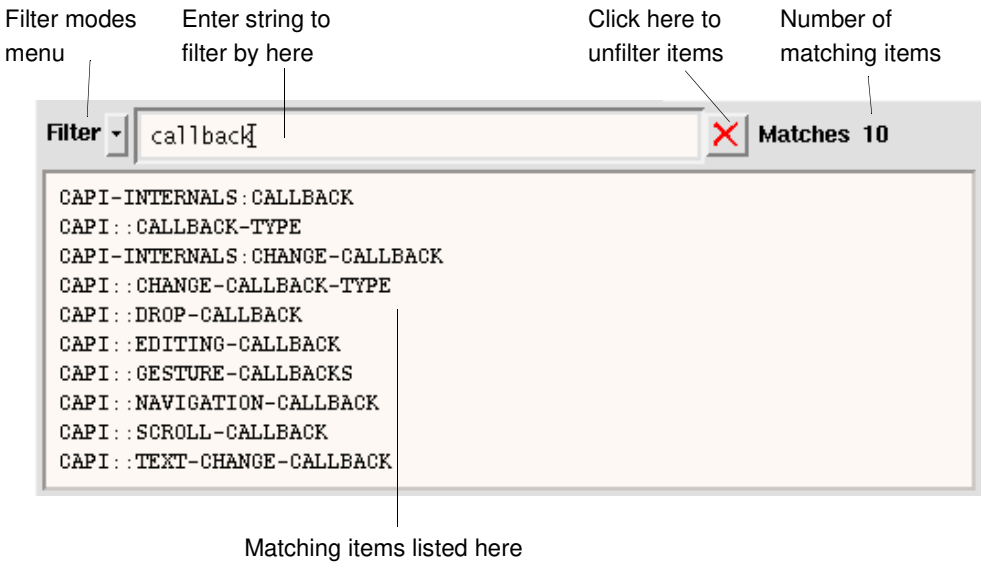
Such lists are often long, including information which you are not interested in. For instance, Common Lisp objects may contain a large number of slots, most of which are of no importance to your work.

Most such lists in Common LispWorks have a filter area which allows you to hide the uninteresting information. The filter area is above the list, and consists of the Filter pane into which you can enter text, toolbar buttons, and the Matches pane. There is also a filter modes dropdown menu, described in “Advanced Filtering” on page 43.

3.11.1 Plain Filtering

This section describes how you can filter list items based on a substring match.

Figure 3.6 Filter area with plain match



To use the Filter box, simply enter text. The list is filtered automatically as you type. Only those items that contain the specified string are displayed in the list—all the others are hidden from the display. The number of items that are listed is printed in the Matches area to the right of the Filter box.

To unfilter items (that is, display all the items in a list once again) delete the string in the Filter box or click the button.

3.11.2 Advanced Filtering

This section describes how you can filter list items by a regular expression match rather than a plain string match, make the match case-sensitive, and how to invert the filter.

To alter the way that the filter operates, select one or more options from the **Filter** dropdown menu to the left of the filter pane. This can be done using the mouse, but is more convenient with a keyboard gesture. Each gesture selects or deselects one filter mode. The keyboard gestures invoking advanced filter modes are shown in Table 3.2.

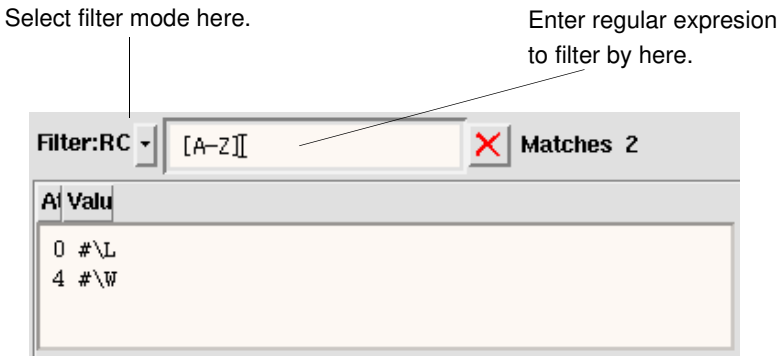
Table 3.2 Advanced Filter modes

Keyboard gesture	Filter mode	Description
`Ctrl+Shift+R`	**Regexp Search**	Filters by regular expression matching
`Ctrl+Shift+E`	**Exclude Matches**	Excludes items matching the filter
`Ctrl+Shift+C`	**Case Sensitive**	Filters by a case-sensitive comparison

The choice of items displayed changes according to the content of the filter pane and the selected filter options. The label on the **Filter** dropdown changes to indicate your selected filter options.

In the example illustrated below, we have inspected the string **"LispWorks"**, entered a regular expression which matches alphabetic characters, and pressed **Ctrl+Shift+R Ctrl+Shift+C** to select the **Regexp Search** and **Case Sensitive** filter modes.

Figure 3.7 Filter area with regular expression match



Now press **Ctrl+Shift+E** to select the **Exclude Matches** filter option. Only the lowercase characters of the string **"LispWorks"** are displayed in the list.

Note: For details of the regular expression syntax, see “Regular expression syntax” on page 44.

Note: The three filter modes are mutually independent.

3.12 Regexp matching

Regular expressions (regexps) can be used when searching and filtering throughout the IDE. This section describes exactly how LispWorks regexp matching operates.

3.12.1 Regular expression syntax

. Matches any single character except a newline. For example, **c.r** matches any three character string starting with **c** and ending with **r**.

***** Matches the previous regexp any number of times (including zero times). For example, **ca*r** matches strings beginning with **c** and ending with **r**, with any number of **a**s in-between.

`+`	Matches the previous regexp any number of times, but at least once. For example, `ca+r` matches strings beginning with `c` and ending with `r`, with at least one `a` in-between.
`?`	Matches the previous regexp either 0 or 1 times. For example, `ca?r` matches either the string `cr` or `car`, and nothing else.
`^`	Matches the next regexp as long as it is at the beginning of a line. For example, `^foo` matches the string `foo` as long as it is at the beginning of a line.
`$`	Matches the previous regexp as long as it is at the end of a line. For example, `foo$` matches the string `foo` as long as it is at the end of a line.
`[ ]`	Contains a character set to be used for matching, where the other special characters mentioned do not apply. The empty string is automatically part of the character set. For example, `[a.b]` matches either `a` or `.` or `b` or the empty string. The regexp `c[ad]*r` matches strings beginning with `c` and ending with `r` , with any number of `a`s and `d`s in-between. The characters `-` and `^` have special meanings inside character sets. `-` defines a range and `^` defines a complement character set. For example, `[a-d]` matches any character in the range `a` to `d` inclusive, and `[^ab]` matches any character except `a` or `b`.
`\`	Quotes the special characters. For example, `\*` matches the character `*` (that is, `*` has lost its special meaning).
`|`	Specifies an alternative. For example, `ab|cd` matches either `ab` or `cd`.
`\( , \)`	Provides a grouping construct. For example, `ab\(cd|ef\)` matches either `abcd` or `abef`.

3.12.2 Regexp and plain string matching

Sometimes you need to select an option to use regexp matching, as the default behavor uses a plain string comparison. For example, see “Advanced Filtering” on page 43.

Other areas always use regexp matching, such as the search target in some modes of the “The Search Files tool” on page 207, and editor commands with names containing **"Regexp"**.

3.13 Completion

Where there is a finite set of meaningful text inputs (symbol names, names of existing files or editor commands, and so on) the IDE helps you to enter your text by offering *completion*. When you invoke completion, the system takes your partial input and either:

- extends your partial input to an unambiguous longer (but possibly partial) input, or
- presents a choice of the possible meaningful inputs.

When your input remains partial, you may repeat the completion gesture.

When you see a choice of the possible meaningful inputs, certain gestures allow you to narrow the choice and quickly select the desired input, as described in “Selecting the completed input” on page 47.

3.13.1 Invoking completion

When a command prompts for input in the echo area, the keys **Tab**, **?** and **Space** can invoke completion, depending on the context.

In the Editor tool, a variety of completion commands are available. For example, in Emacs emulation **Meta+Ctrl+I** invokes the command **Complete Symbol**. See the *LispWorks Editor User Guide* for details of this and other editor commands.

In the Shell tool, **Tab** expands filenames.

In the Listener tool using Emacs emulation, **Escape Tab** expands filenames.

In many text input panes such as the **Class:** field of a Class Browser tool, **Up** and **Down** invoke in-place completion while pressing the button raises a completion dialog.

Also, clicking the button to the right of a text input pane raises a modal completion dialog, as described in “Completion dialog” on page 49.

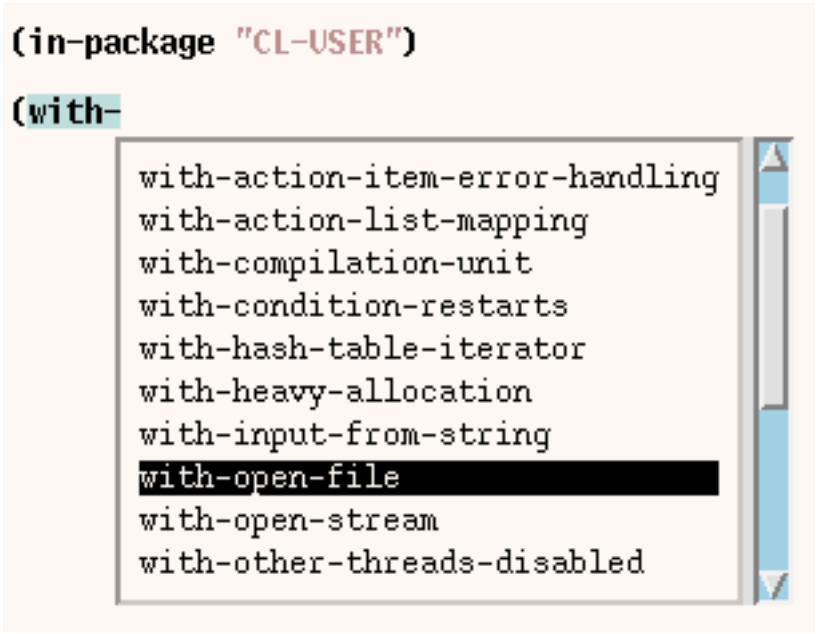
3.13.2 Selecting the completed input

The IDE presents the choice of inputs in one of two ways, described in the next two sections. The option **Works > Tools > Global Preferences... > Use in-place completion** controls whether in-place completion is used.

3.13.2.1 In-place completion

In-place completion presents the choice of complete inputs in a special non-modal window. Figure 3.8 below shows this in the context of the editor command **`Complete Symbol`**.

Figure 3.8 Example in-place completion window



While this window is visible, most keyboard gestures such as unmodified alphanumeric and punctuation keys are processed as ordinary input, adding to your

partial input. This reduces the number of possible completions. Conversely, deleting part of your input will increase the number of possible completions.

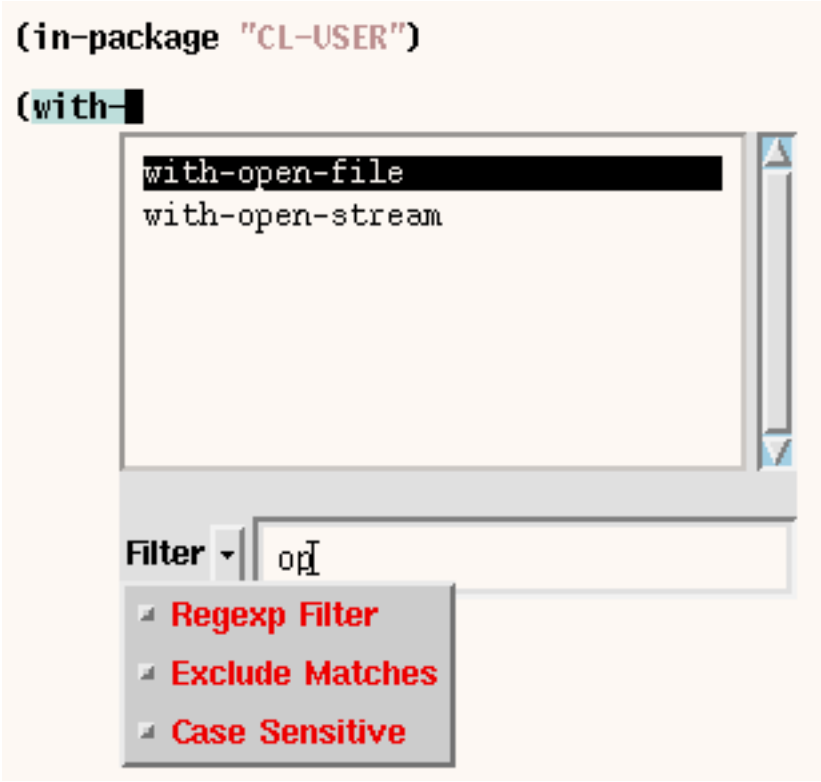
You can navigate the choice with **Up** and **Down** and you can select the desired completion at any time with **Return** or double-click. To cancel the attempt to complete, press **Escape**.

3.13.2.2 Filtering in-place completion

You can reduce the number of displayed completions by adding a filter to the in-place completion window.

To add the filter, press **Ctrl+Return**. To use the filter, type a substring of the desired result. By default, filtering is by a case-insensitive substring comparision.

Figure 3.9 Example in-place completion window with filter



You can set filter modes to alter the way that the filter operates, just as described in “Advanced Filtering” on page 43. Briefly, you select options from the **Filter** dropdown menu or with the keyboard gestures **`Ctrl+Shift+R`**, **`Ctrl+Shift+E`** and **`Ctrl+Shift+C`**. The choice of items displayed changes according to the content of the filter pane and the selected filter options, and the label on the **Filter** dropdown changes to indicate your selected filter options.

3.13.3 Completion dialog

When the **Use in-place completion** option (see “Selecting the completed input” on page 47) is off, all keyboard completion gestures raise a modal dialog presenting a choice of completion options.

Also, clicking the button to the right of a text input pane raises a modal completion dialog.

You can navigate the choice with **`Up`** and **`Down`** and you can select the desired completion at any time with **`Return`**, double-click, or press the **OK** button. To cancel the attempt to complete, press **`Escape`**.

3.13.3.1 Filtering modal dialog completion

A modal completion dialog automatically has a filter if the number of possible completions exceeds the value of the option **Works > Tools > Preferences... > General > Add a filter to dialog lists longer than:**. By default this option has value 25.

The filter options described above are also available in a modal completion dialog, and are controlled by the same keyboard gestures, for example **`Ctrl+Shift+R`**. See “Advanced Filtering” on page 43 for details.

3.14 Examining a window

You can examine any tool window with the **Works > Interface** menu.

This menu contains the standard action commands described in Section 3.7 on page 35. Thus, choose **Works > Interface > Inspect** to inspect the **`capi:interface`** object for the window.

Choose **Works > Interface > Browse - Window** to browse the structure of the window object. From here you can browse the child windows and their CAPI representations.

For information about the tools mentioned, see Chapter 7, “The Class Browser” and Chapter 16, “The Inspector” and Chapter 27, “The Window Browser”.

4

Getting Help

All tools contain a **Help** menu that gives you access to a variety of forms of online help. This chapter describes how to use this online help.

4.1 Online manuals in HTML format

A complete documentation set is provided with LispWorks in the form of HTML files. Asuming that you have installed the documentation, these files are in the directory which is the result of evaluating this form:

```
(sys:lispworks-dir "manual/online/web/")
```

Common LispWorks can link directly to the HTML files provided, allowing you to go straight to the most relevant documentation for the current context.

No proprietary extensions to HTML have been used, so you can use any HTML browser to view the documents. The Common LispWorks **Help** menu is drives the following browsers: Netscape, Firefox, Mozilla, Opera .

4.1.1 Browsing manuals online

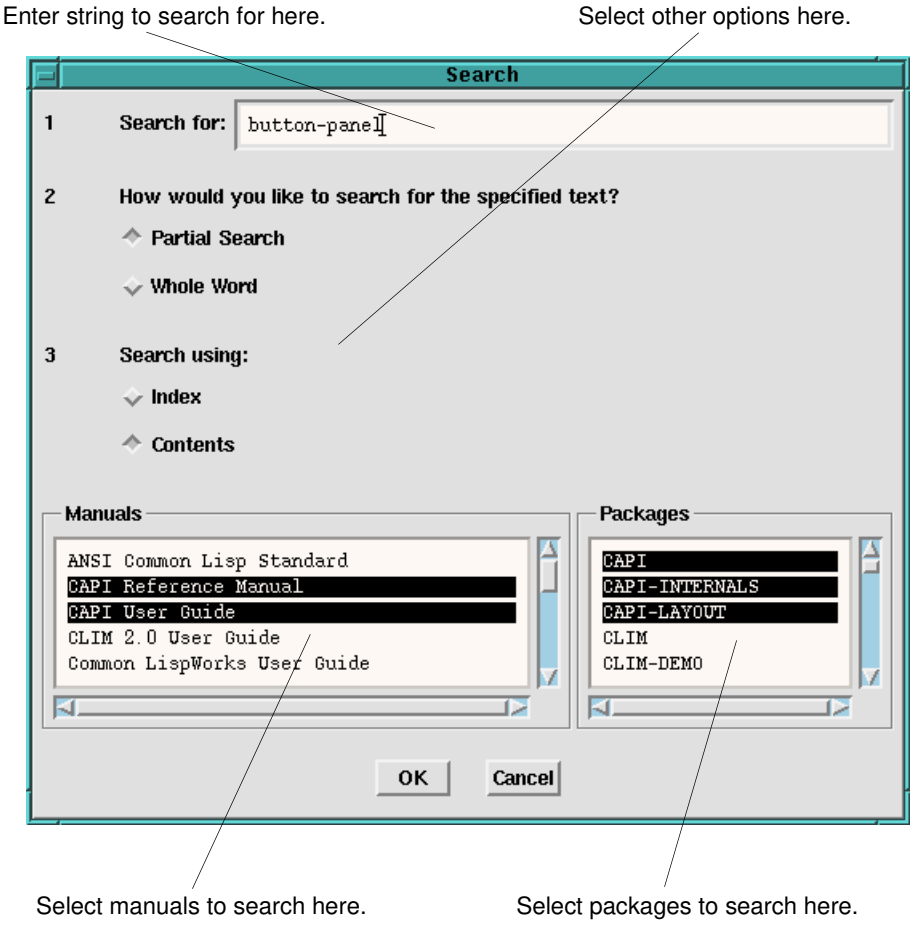
Choose **Help > Manuals...** to browse any of the available manuals online. A dialog appears, allowing you to choose a manual.

If you already have an HTML browser running, a link to the first page of the manual you choose is displayed in it. If you do not have a browser running, one is started for you.

4.1.2 Searching the online manuals

Choose **Help > Search...** to search the online documentation. The Search dialog, shown in Figure 4.1, appears.

Figure 4.1 Search dialog



This dialog lets you specify what you want to search for, and which manuals you want to search in.

Enter a string of text in the **Search for** area.

There are a number of additional options that you can set if you want:

- Select **Whole Word** if you want to confine your search to whole words only. Select **Partial Search** if you want to match part of a word as well. By default, partial searches are performed. For example, if **Whole Word** is selected, searching for “pane” only matches the word “pane”. If **Partial Search** is selected, searching for “pane” also matches “panels”.
- You can choose whether to search the index or the table of contents of any given manual; select **Index** or **Contents** as appropriate. By default, indexes are searched, as these tend to produce the richest information.

Select the manuals you want to search in the Manuals list. If nothing is selected, all manuals are searched. You can select any number of items in this list.

Select the packages you want to search from the Packages list. If nothing is selected (the default), all packages are searched. You can select any number of items in this list.

Note that selections made in the Manuals and Packages lists reflect each other. If you choose one or more manuals, the relevant packages are also selected, and if you choose one or more packages, the relevant manuals are selected.

Once you have specified the search options, click **OK**. The results of the search are displayed in your HTML browser.

The **Help** menu has a command **Lisp Knowledgebase...** which takes you to the LispWorks support page where a knowledgebase of problems and solutions can be searched. There is also a command **LispWorks Patches...** which takes you to the page where you can download the latest patches for LispWorks. The support page also contains guidelines on how to prepare a useful bug report.

4.1.3 Getting help on the current tool

Choose **Help > On Tool...** to get help on the current tool. This takes you to the appropriate online chapter of this manual.

4.1.4 Getting help on the current symbol

Choose **Help > On Symbol...** to search for help on the symbol under the point (in an editor-based window) or the current object of a tool. This option displays the Search dialog described in Section 4.1.2, but with options pre-selected to enable you to search for documentation on the current symbol. Click **OK**, and the results of the search are displayed in your HTML browser.

4.1.5 Getting help from the LispWorks website

Please check our online knowledgebase at **`www.lispworks.com`** before reporting problems to Lisp Support. You may find a solution or workaround there. Choose **Help > Lisp Knowledgebase** to visit the online knowledgebase.

4.1.6 Getting patches from the LispWorks website

You must run LispWorks with the latest patch release installed. This is available from **`www.lispworks.com`**. The command **Help > LispWorks Patches** is a convenient shortcut to visit the patch download page.

4.1.7 Configuring the browser used

We recommend that you use Netscape. You can specify the location of the browser used by **Help > Browser Preferences... > Browser > Browser Executable Location**. You can enter the directory here in the Directory window. However, the default setting, **Use PATH**, is adequate for most users. It means that the Netscape executable found via your UNIX environment variable PATH is used.

Alternatively, set the variable **`*browser-location*`** (details in the *LispWorks Reference Manual*).

4.2 Online help for editor commands

You can display online help for any available editor command using the commands under **Help > Editing**. See Section 12.14 on page 185 for details.

4.3 Browsing manuals online using Adobe

Reader

The LispWorks manuals are also available in PDF (Portable Document Format). These can be found in the LispWorks library directory **lib/5-1-0-0/manual/offline/pdf** .

You can view these files online and print them using Adobe Reader, which can be downloaded freely from the Adobe web site at **www.adobe.com**.

Note: PDF format documentation is not supplied with LispWorks Personal Edition, in order to minimize the size of the download. However, you may download the PDF format manuals at any time from the LispWorks website at **www.lispworks.com/reference/**.

4.4 Reporting bugs

The **Help** menu also contains a **Report Bug** command that provides a template for reporting LispWorks bugs. Please do use this when you contact Lisp Support.

Before sending a report, please check the instructions at **www.lispworks.com/support/**.

5

Manipulating Graphs

Views that use graphs are provided in the Class Browser, Function Call Browser, and Window Browser. These views let you, for instance, produce a graph of all the subclasses or superclasses of a given class, or the layouts of a given CAPI interface.

In the Class Browser, the subclasses and superclasses views use graphs. The Function Call Browser uses graph views for its Called By and Calls Into views. There is only one view in the Window Browser, and that uses a graph.

All graphs in Common LispWorks can be manipulated in the same way. This chapter gives you a complete description of the features available.

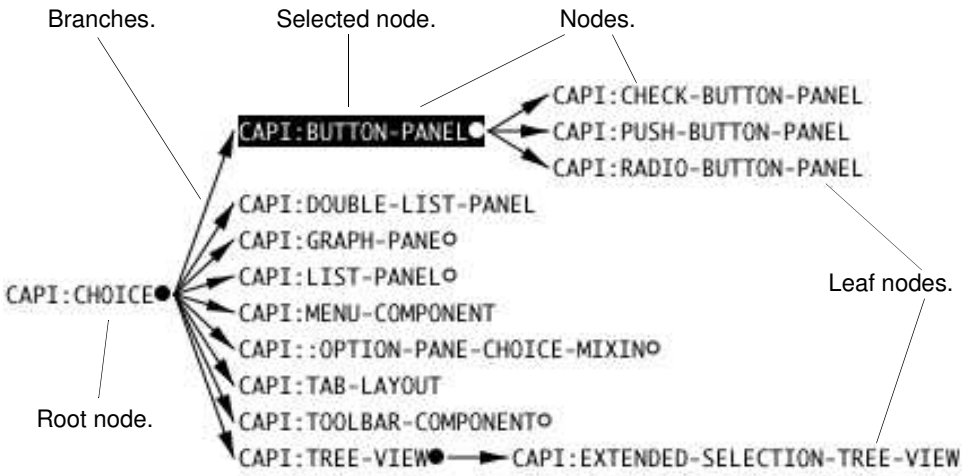
All graphs have an associated graph layout menu, available by displaying a context menu over the graph itself by using the alternate select gesture. This menu contains all the commands that are directly relevant to graphs.

5.1 An overview of graphs

An example graph is shown in Figure 5.1 below. All graphs are laid out by Common LispWorks, so that their elements are displayed in an intuitive and easily visible hierarchy. A graph consists of a number of *nodes*, linked together by *branches*. By default, graphs in the environment are plotted from left to right: for any given node, the node to which it is linked on the left is known as its *parent*, and the nodes to which it is linked on the right are known as its *children*. The originating node of the graph (on the

far left) is referred to as the *root node*, and the outermost nodes of the graph (towards the right) are referred to as *leaf nodes*. The root node does not have a parent, and leaf nodes do not have any children.

Figure 5.1 Example graph pane



You can select nodes in a graph pane in exactly the same way that you select items in a list. Selected nodes are highlighted, as shown in Figure 5.1.

Similarly, you can copy nodes from a graph onto the clipboard in a manner consistent with use of the clipboard in the rest of the environment. When you copy any selected node onto the clipboard, the Lisp object itself is copied onto the clipboard, so that it can be transferred into other Common LispWorks tools.

UNIX Implementation Note: The string representation of the Lisp object is copied into the UNIX clipboard, so that it can be transferred to other applications.

5.2 Searching graphs

Sometimes graphs can be too large to fit onto the screen at once. In this case, it is useful to be able to search the graph for any nodes you are interested in. There are two commands which let you do this.

Choose **Edit > Find...** to find any node in the graph whose name contains a given string. Choose **Edit > Find Next** to find the next node in the graph that contains that

string. Whenever a matching node is found, it is selected in the graph. If necessary, the window scrolls so that the selected node is visible.

Note that you do not have to specify a complete node name: to find all nodes that include the word “debug” in their name, just type **`debug`** into the dialog. All searches are case insensitive.

A full description of these commands can be found in Section 3.3.4 on page 30.

5.3 Expanding and collapsing graphs

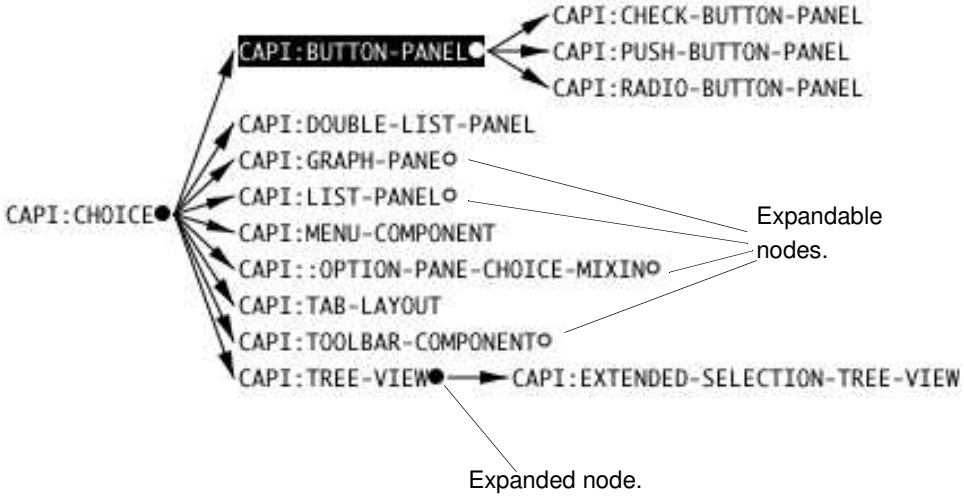
You may often find that you are only interested in certain nodes of a graph. Other nodes may be of no interest and it is useful, especially in large graphs, to be able to remove their children from the display.

Notice that some nodes have a small circle drawn alongside them, as shown in Figure 5.2. The circle indicates that the node is not a leaf node, that is, it has children. Moreover, the circle is filled black if the node is currently expanded, and is unfilled if the node is currently expandable (also referred to as collapsed).

5.3.1 Expanding and collapsing by clicking

To collapse or expand any node with children in a graph, click on the circle alongside it. Thus, click on the unfilled circle of an expandable node to display its children, and click on the filled circle of an expanded node to hide its children.

Figure 5.2 Expanded and expandable nodes



For instance, in Figure 5.2, click on the unfilled circle alongside CAPI:TOOLBAR-COMPONENT to display its subclasses. Click on the filled circle to hide them.

5.3.2 Expanding and collapsing by menu commands

You can also collapse or expand nodes using the context menu:

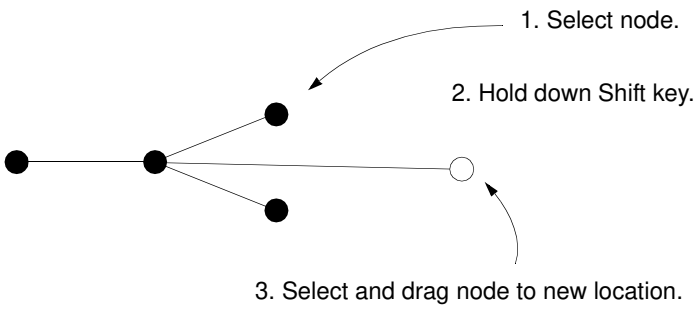
- Choose **Expand Nodes** to expand the selected node.
- Choose **Collapse Nodes** to collapse the selected node.

5.4 Moving nodes in graphs

Although the layout of any graph is calculated automatically, you can move any node in a graph manually. This can be useful if the information in the graph is dense enough that some nodes are overlapping others.

To move the selected node, hold down the **Shift** key and select and drag the node to the desired location.

Figure 5.3 Moving a node in a graph



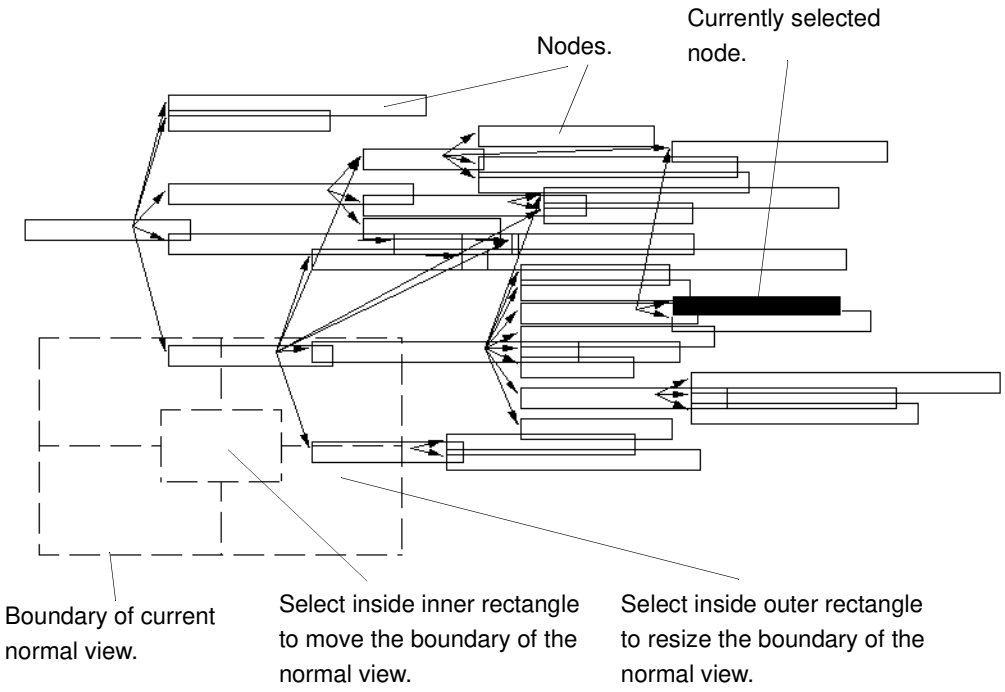
At any time, you can choose **Reset Graph Layout** from the context menu to restore the nodes to their original positions.

5.5 Displaying plans of graphs

Many graphs are too large to be able to display in their entirety on the screen. As with any other window, you can use the scroll bars to display hidden parts of the graph. However, you can also display a plan view of the entire graph.

To display the plan view of any graph, hold down the **`Control`** key and select the graph, or choose **Enter Plan Mode** from the context menu. The graph is replaced by its plan view, similar to the one shown in Figure 5.4.

Figure 5.4 Example plan view



Each node in the original graph is represented by a rectangle in the plan view. The currently selected node is shown as a filled rectangle, and all other nodes are clear. You can select nodes in the plan view, just as you can in the normal view.

A dotted grid is drawn over the plan view; you can use this grid to alter the section of the graph that is shown in the normal view. The size and position of the grid represents the portion of the graph that is currently displayed in the normal view.

- To move the grid, so that a different part of the graph is shown in the normal view, hold down **`Shift`** and select and drag the innermost rectangle of the grid. The entire grid moves with the mouse pointer.

- To resize the grid, so that a different proportion of the graph is shown, hold down **`Shift`** and select and drag the outermost rectangle of the grid. The entire grid will resize. You can select any part of the grid except the innermost rectangle to perform this action.

To return to the normal view, hold down **`Control`** and select the graph again, or choose **Exit Plan Mode** from the context menu. The part of the graph indicated by the grid in the plan view is displayed.

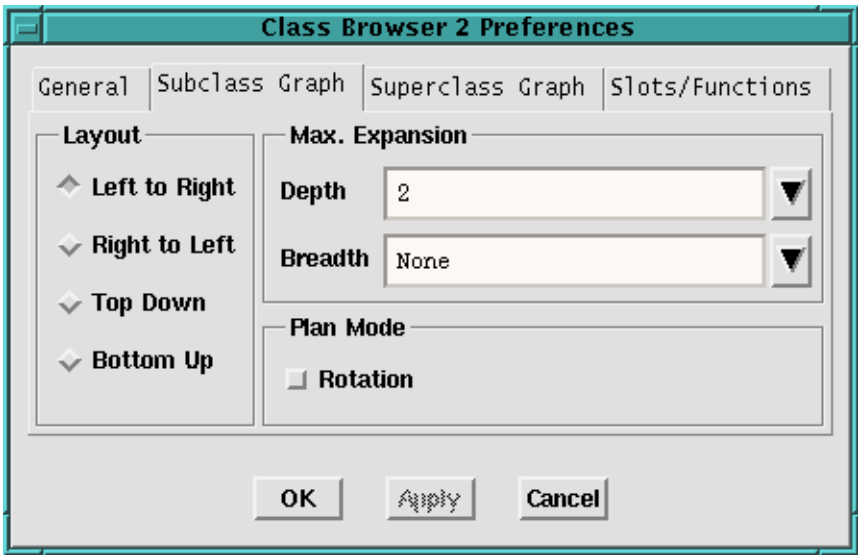
5.6 Preferences for graphs

A number of graph layout preferences can be set for any tool that uses graphs. These preference settings are available in the Preferences dialog for the tool, and can be displayed by doing either of the following:

- Choose **Preferences...** from the graph layout context menu.
- Choose **Works > Tools > Preferences...** from the relevant tool, and click on a graph layout tab in the Preferences dialog.

For example, the graph layout preferences for the Class Browser are shown in Figure 5.5.

Figure 5.5 Graph layout preferences



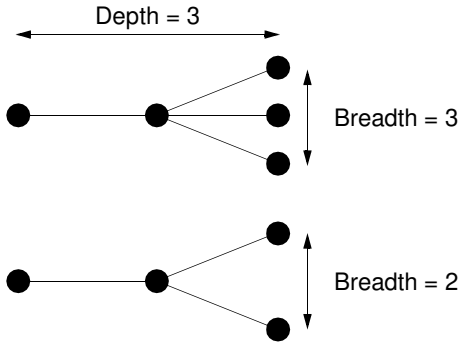
This section describes the options available in the graph layout tabs of the Preference dialogs for any tool that uses graphs.

5.6.1 Altering the depth and breadth of graphs

For large graphs, you may find that you want to alter the maximum depth and breadth in order to simplify the information shown. Each graph pane has its own depth and breadth setting, which is used for all graphs drawn in it. These are available in the **Max Expansion** panel of the graph layout tabs in the Preferences dialog.

The depth and breadth of a graph are depicted in Figure 5.6.

Figure 5.6 Depth and breadth of graphs



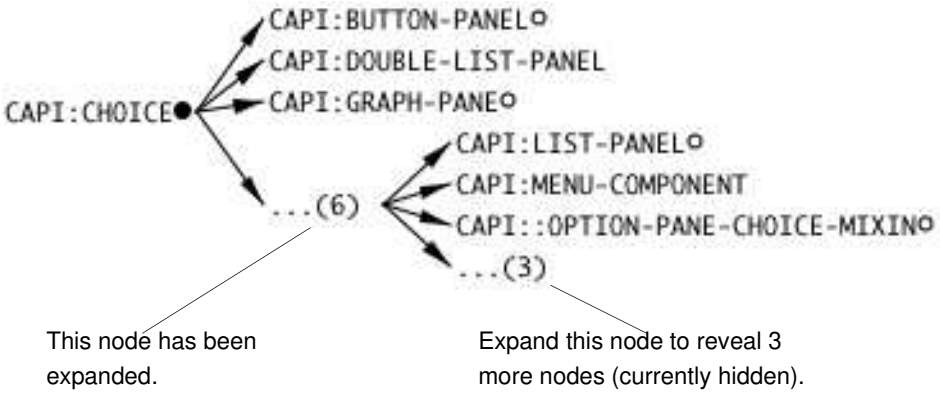
Choose a number from the Depth list to change the maximum depth of graphs in a given tool. The depth of a graph is the number of generations of node which are displayed. Most graphs have a default initial depth of 2, which means that you must expand any nodes you want to investigate by expanding them yourself. The default value is 2.

Note that the maximum depth setting is ignored for nodes which you have expanded or collapsed. See Section 5.3 on page 59.

Choose a number from the Breadth list to change the maximum breadth of a given tool. The breadth of a graph is the number of child nodes which are displayed for each parent. If there are more children than can be displayed (the maximum breadth setting is less than the number of children for a given node) an extra node is visible. This node is labeled "**...**", followed by the number of nodes that remain undisplayed. Nonetheless you can expand this node by the **Expand Nodes** command allowing you to display the additional children without having to alter the maximum breadth setting for the whole graph. By default, the maximum breadth is set to None, so that all the

children for a node are displayed, no matter how many there are. An example of this feature is shown in Figure 5.7 below, where the maximum breadth has been set to 3.

Figure 5.7 Displaying children hidden by the maximum breadth setting



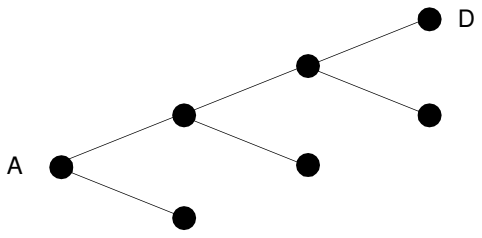
To ensure that all available information is graphed in a given tool, set both the maximum depth and maximum breadth to None.

5.6.2 Displaying different graph layouts

As already mentioned, graphs are laid out from left to right by default, but they can be laid out in other orientations. This can be configured in the Layout panel of the graph layout tab in the Preferences dialog.

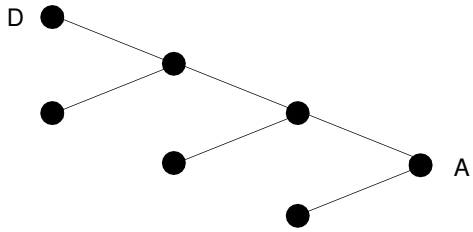
Click “Left to Right” to layout a graph from the left of the screen to the right, as shown in Figure 5.8. This is the default orientation for every graph in the environment.

Figure 5.8 Left to right layout



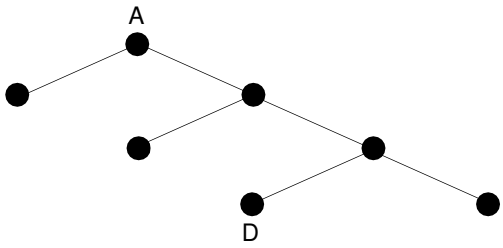
Click “Right to Left” to layout a graph from the right of the screen to the left, as shown in Figure 5.9.

Figure 5.9 Right to left layout



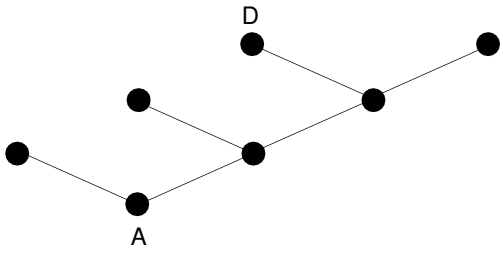
Click “Top Down” to layout a graph from the top of the screen to the bottom, as shown in Figure 5.10.

Figure 5.10 Top down layout



Click “Bottom Up” to layout a graph from the bottom of the screen to the top, as shown in Figure 5.11.

Figure 5.11 Bottom up layout



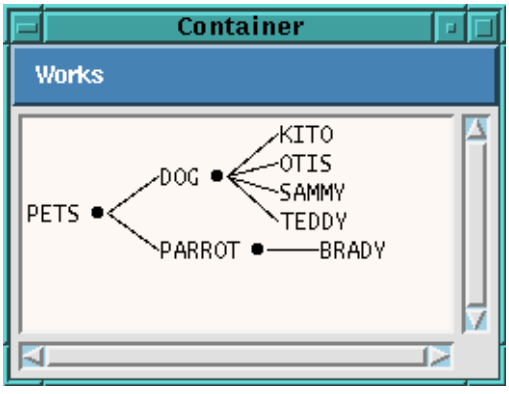
5.7 Using graphs in your programs

You can read about the CAPI class **graph-pane** in the *LispWorks CAPI Reference Manual* for detailed API information for using graphs in your own programs. We will also look at a short example in this section. The following code listing defines a call-back function and creates a graph-pane object:

```
(defun node-children (node)
  (if (equal node 'pets)
      (list 'dog 'parrot)
    (if (equal node 'dog)
        (list 'Kito 'Otis 'Sammy 'Teddy)
      (if (equal node 'parrot)
          (list 'Brady)))))

(setq test-graph
      (capi:contain
       (make-instance 'capi:graph-pane
                      :roots '(pets)
                      :children-function
                      'node-children)
       :best-width 300
       :best-height 400))
```

The children function **node-children** should return **nil** for a leaf node in the graph or a list of child nodes for a non-leaf node. Figure 5.12 shows the generated graph-pane.

Figure 5.12 Sample Graph from a User Program



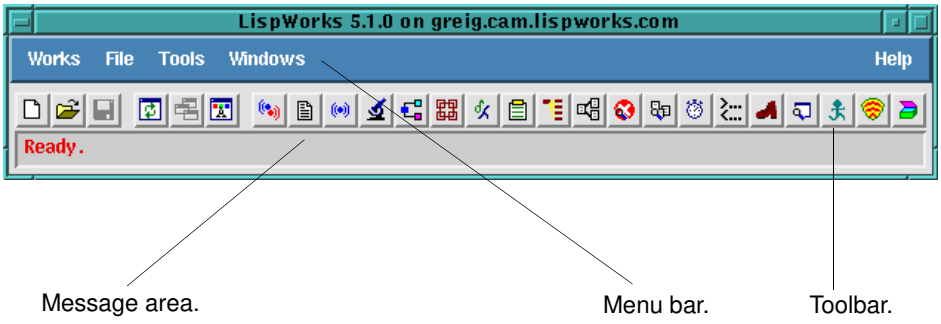
6

The Podium

6.1 The podium window

When you start the environment, by default a window known as the *podium* appears.

Figure 6.1 The LispWorks podium



The podium contains a menu bar, a toolbar, and a message areaThe icons in the podium's toolbar access the Listener, Editor, Output Browser, Inspector, Class Browser, Generic Function Browser, Symbol Browser, Object Clipboard, Function Call Browser, System Browser, Compilation Conditions Browser, Search Files, Profiler, Tracer, Stepper, Window Browser, Process Browser, Shell and Application Builder tools. If you hold the mouse over these icons for a second, the coresponding tool name will appear as floating help text.

The IDE tools have most of these menu items in common with the podium.

The menu bar contains five menus:

- The **Works** menu contains commands that operate on the current window.
- The **File** menu contains commands that load, save and compile Lisp files.
- The **Tools** menu contains commands to create and configure Common LispWorks tools.
- The **Windows** menu lists all the current windows in the environment. To make any window the active window, choose it from this menu.
- The **Help** menu contains commands described in Chapter 4, “Getting Help”.

6.2 Specifying the initial tools

By default the LispWorks IDE starts up with the Podium and a Listener.

If you want to see other tools each time you start Common LispWorks, then you can add action items in your personal initalization file, or in a saved image.

For example, to start an Editor tool, define an action on the pre-defined "Initialize LispWorks tools" action-list:

```
(define-action "Initialize LispWorks tools"
               "Make an Editor Tool"
               #'(lambda (screen)
                   (capi:find-interface 'lw-tools:editor
                                        :screen screen))
               :after "Create default the tools")
```

Note: the names of the various tools are exported in the **LW-TOOLS** package.

For more information about action lists, including an example which opens specific files in the Editor tool, see the *LispWorks User Guide*.

7

The Class Browser

The Class Browser allows you to examine Common Lisp classes. It contains seven views, allowing you to view class information in a number of different ways. You can display each view by clicking the appropriate tab. The available views are as follows:

- The slots view is used to look at the slots available to the class browsed. This view is rich in information, showing you details about items such as the readers and writers of the selected slot.
- The subclasses view produces a graph of the subclasses of the current class, giving you an easy way to see the relationship between different classes in the environment.
- The superclasses view produces a graph of the superclasses of the current class, giving you an easy way to see the relationship between different classes in the environment.
- The hierarchy view lets you see the immediate superclasses and the immediate subclasses of the current class, using a text-based interface.
- The initargs view allows you to see the initargs of the current class together with information about each initarg. See Section 7.6 on page 91 for more details on how you can use this view.

- The functions view allows you to see information about the CLOS methods that have been defined on the current class. See Section 7.5 on page 88 for details on using the information in this view.
- The precedence view is used to show the class precedence list for the current class. See Section 7.7 on page 93 for more details on how you can use this information.

To create a Class Browser, choose **Tools > Class Browser**. Alternatively, use **`Meta+X Describe Class`** in an Editor or choose **Class** from any submenu that provides the standard action commands to invoke a Class Browser on the Lisp object referred to by that submenu. This automatically browses the class of the Lisp object. For more information on how the standard action commands refer to objects in the environment, see Section 3.7 on page 35.

7.1 Simple use of the Class Browser

This section describes some of the basic ways in which you can use the Class Browser by giving some examples. If you wish, you can skip this section and look at the descriptions of each individual view: these start with Section 7.2 on page 79.

When examining a class, the slot names of the class are displayed by default.

To examine a class, follow the instructions below:

1. Create a push button panel by entering the following in the Listener:

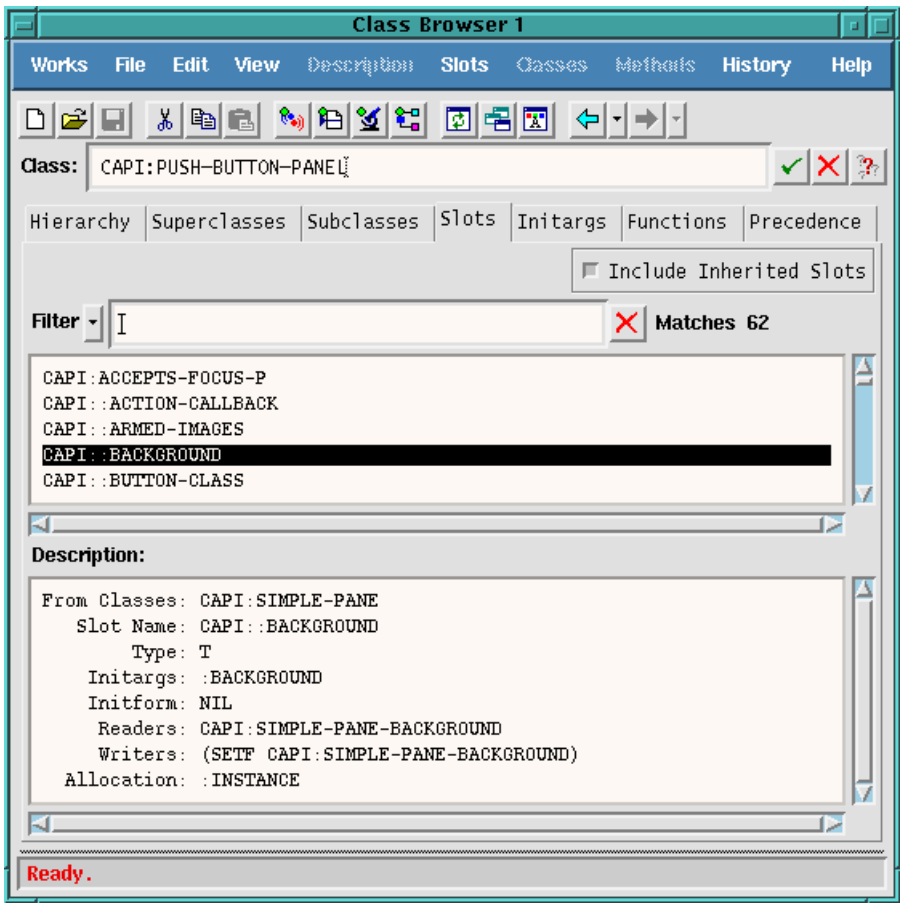
```
(capi:contain
 (make-instance 'capi:push-button-panel
                :title "Test Buttons"
                :items '(:one :two :three)))
```

   The push button panel appears on your screen.

2. With the Listener as the active window, choose **Values > Class**.

This invokes the Class Browser on the button panel. The class **`capi:push-button-panel`** is described in the Class Browser.

Figure 7.1 Examining classes in the Class Browser



Notice that, although you invoked the browser on an object that is an instance of a class, the class itself is described in the Class Browser. Similarly, if you had pasted the object into an Inspector, the instance of that object would be inspected. Using the environment, it is very easy to pass Common Lisp objects between different tools in this intelligent fashion. This behavior is achieved using the Common LispWorks clipboard; see Section 3.3.2 on page 28 for details.

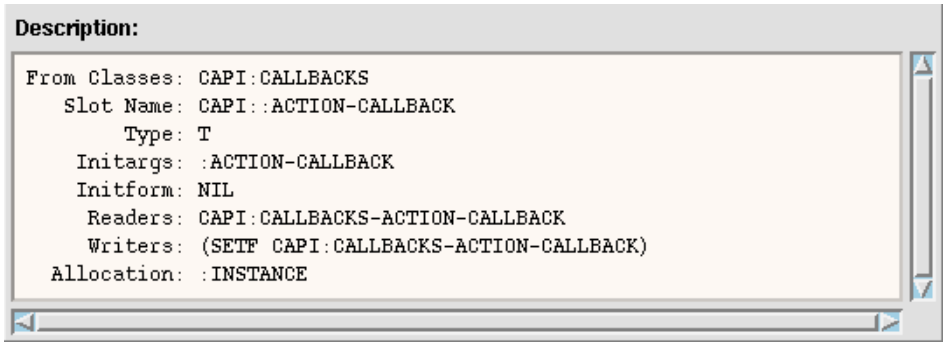
See Section 3.7 on page 35 for a full description of the standard action commands available.

7.1.1 Examining slots

A list of the slots in the current class is printed in the Slots area. By selecting any slot, you can examine it in more detail in the Description area.

While still examining the **`capi:push-button-panel`** class, select any slot in the Slots area.

Figure 7.2 Description of a slot



A description of the slot is given in the Description area. For details about the information contained in this description, see Section 7.2.4 on page 80.

7.1.2 Examining inherited slots

By default, inherited slots (those slots which are defined in a superclass of the current class, rather than the current class itself) are listed in the Slots area along with the slots defined in the current class. Deselect the **Include Inherited Slots** button just above the Filter box to inhibit this listing.

1. While still examining the **`capi:push-button-panel`** class, click **Include Inherited Slots** to deselect this option.

No slots are displayed in the Slots area. This is because all the slots available to the **`capi:push-button-panel`** class are inherited from its superclasses. No slots are defined explicitly on the **`capi:push-button-panel`** class.

2. Select **Include Inherited Slots** again, and then select a few slots in the Slot area in turn.

Notice that the slot description for each slot tells you which superclass the slot is defined on.

7.1.3 Filtering slot information

The Filter box can be used to filter out information about slots you are not interested in. This is especially useful if you are examining classes which contain a large number of slots.

The example below shows you how to create an instance of a CAPI object, and then limit the display in the Class Browser so that the only slots displayed are those you are interested in:

1. In a Listener, create a button object by typing the following:

```
(capi:contain (make-instance 'capi:list-panel
                             :items '("Apple" "Orange" "Pear")))
```

   This creates a list panel object and displays it on your screen. The list panel object is the current value in the Listener.

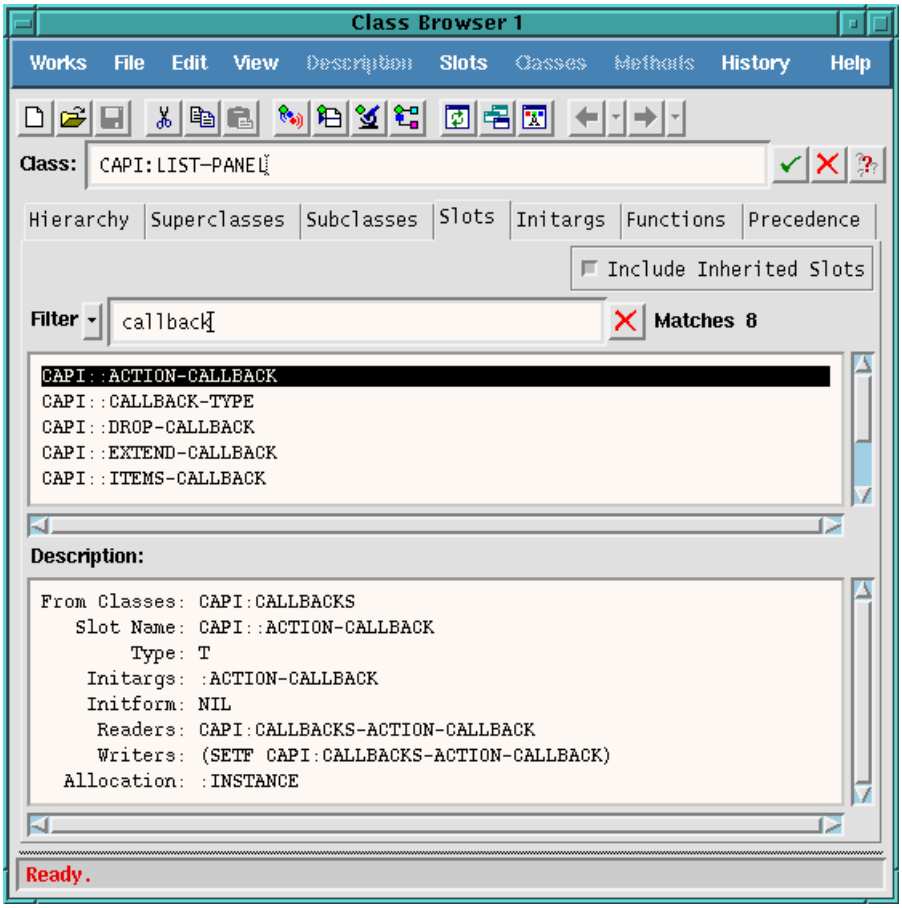
2. Make the Listener window active and choose **Values > Class** to examine the class of the object in the Class Browser.

3. Click the Slots tab in the Class Browser to switch to the Slots view.

   Suppose you are only interested in seeing the callbacks that can be defined in a list panel.

4. Type **`callback`** in the Filter box.

Figure 7.3 Using filters to limit the display in the Class Browser



You can immediately see the types of callback that are available to CAPI list panel objects. See the *LispWorks CAPI Reference Manual* for details about these callbacks.

For more information about using filters, see “Filtering information” on page 42.

7.1.4 Examining other classes

There are two ways that you can examine other classes. The first is to type the name of the class you wish to see into the Class text box at the top of the browser. Press **Return** or click ✓ and the named class is described.

1. While still examining class **capi:list-panel**, type **capi:push-button-panel** into the Class area.

The class **capi:push-button-panel** is described.

Because some class names may be potentially quite long, you can use *completion* to reduce typing. This allows you to select from a list of all class names which begin with the partial input you have entered. See "Completion" on page 46 for detailed instructions. When you have entered the complete class name, click on ✓ to make this the class being described.

The second way to examine other classes is by using the Superclasses and Subclasses lists available in the hierarchy view. Click on the Hierarchy tab to display the hierarchy view.

The main part of the hierarchy view consists of two lists:

- The Superclasses list shows all the superclasses of the current class.
- The Subclasses list shows all the subclasses of the current class.

Double-click on any superclass or subclass of the current class to examine it.

1. Double-click on **CAPI:BUTTON-PANEL** in the Superclasses list.

   The **capi:button-panel** class is described.

2. Double-click on **CAPI:PUSH-BUTTON-PANEL** in the Subclasses list.

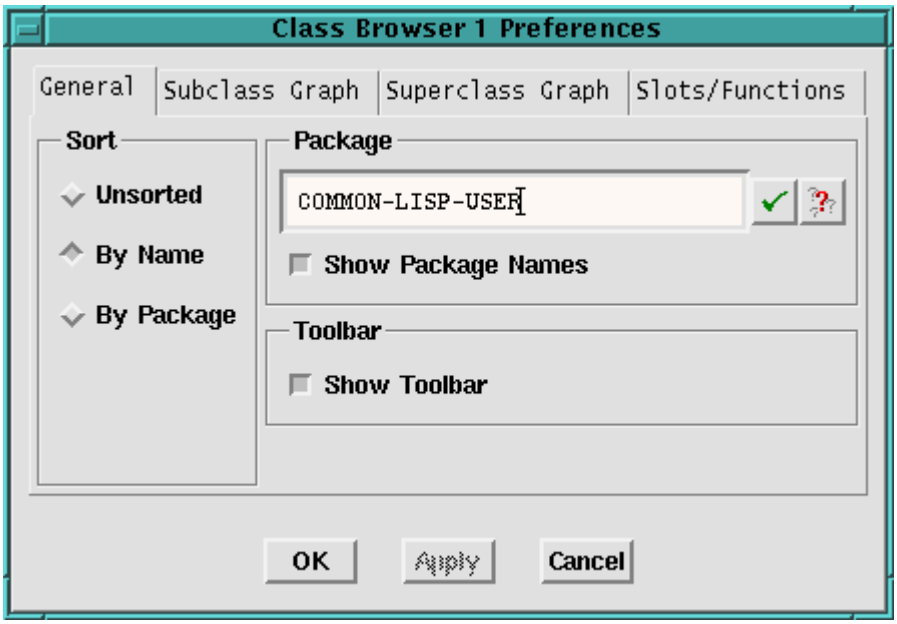
   The **capi:push-button-panel** class is described again.

So, using the Hierachy tab, you can easily look through the related classes in a system.

7.1.5 Sorting information

As with many of the other tools in Common LispWorks, you can sort the items in any of the lists or graphs of the Class Browser using the Preferences dialog. Choose **Works > Tools > Preferences...** or click in the Class Browser to display this dialog.

Figure 7.4 Setting Class Browser preferences



Under the General tab, there are three options for sorting items, listed in the Sort panel.

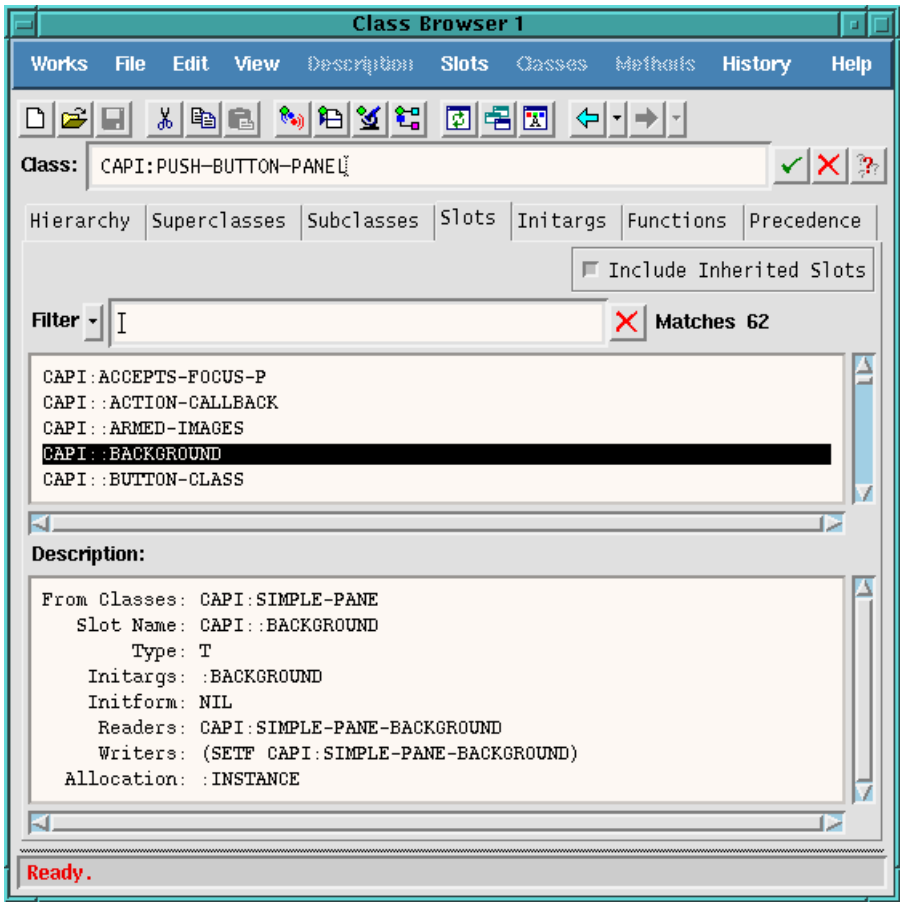
- Unsorted - Displays items in the order they are defined in.
- By Name - Sorts items alphabetically by name. This is the default setting.
- By Package - Sorts items alphabetically by package name.

For more information on sorting items, see Section 3.8.1 on page 38.

7.2 Examining slot information

When the Class Browser is first invoked, the default view is the slots view. You can also click the Slots tab to swap to it from another view. The slots view is shown in Figure 7.5.

Figure 7.5 Examining slots in the Class Browser



Section 7.1 on page 72 introduced you to the slots view in the Class Browser. This section gives a complete description of this view. For completeness, some information may be repeated.

The areas available in the slots view are described below.

7.2.1 Class box

You enter the name of the class you want to browse in the Class text box. You can type in a class name explicitly, or you can transfer a class to the Class Browser using the **Class** standard action command in another tool, or by pasting a class in explicitly.

Note: You can use **Edit > Paste** to paste a class name into this area, even if the clipboard currently contains the string representation of the class name, rather than a class object itself. This lets you copy class names from other applications directly into the Class Browser. See Section 3.3.2 on page 28 for a complete description of the way the Common LispWorks clipboard operates, and how it interacts with the UNIX clipboard.

7.2.2 Filter area

The Filter area lets you restrict the information displayed in the Slots list. See "Filtering information" on page 42 for a description of how to use the Filter area in any tool, and Section 7.1.3 on page 75 for an example of how to use it in the Class Browser.

7.2.3 Slots list

The largest section of the slots view lists the slot names of the current class. Selecting a slot in this list displays a description of it in the Description list, and you can operate on any number of selected slots using the commands in the **Slots** menu.

If **Include Inherited Slots** is selected, slots inherited from the superclasses of the current class are listed as well as those explicitly defined on the current class. Deselect this button to see only those slots defined on the current class. In the Slots/Functions tab of the Class Browser Preferences dialog, you can also configure the default setting of this option. Choose **Works > Tools > Preferences...** or click in the Class Browser to display this dialog.

The number of items listed in the Slots area is printed in the Matches box.

7.2.4 Description list

This list displays a description of the selected slot. The following information is printed:

- From Classes - The classes that this slot is defined in.

- Slot Name - The name of the slot.
- Type - The slot type.
- Initargs - The initargs, if any, which can be used to refer to the slot.
- Initform - The initform, or initial value, of the slot.
- Readers - The readers of the slot. These are the names of any functions which can be used to read the current value of the slot.
- Writers - The writers of the slot. These are the **`setf`** methods which may be used to change the slot value.
- Allocation - The allocation of the slot.

To operate on any of the items displayed in this area, select them and choose a command from the **Description** menu. This menu contains the standard action commands described in Section 3.7 on page 35. You can operate on more than one item at once by making multiple selections in this area.

7.2.5 Performing operations on the current class

You can operate on the current class using the commands in the **Classes** menu. The standard action commands described in Section 3.7 on page 35 are available in this submenu.

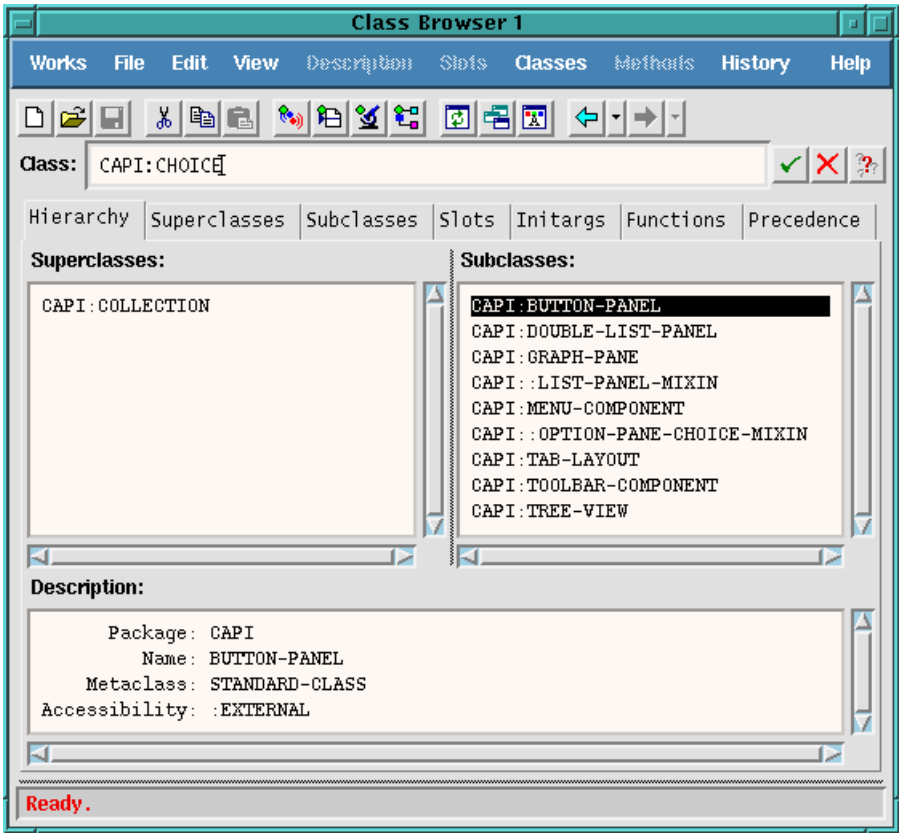
Choose **Classes > Browse Metaclass** to select, and describe in the normal way, the class of the current class.

7.3 Examining superclasses and subclasses

The hierarchy view of the Class Browser lists the immediate superclasses and subclasses of the current class. This view can be useful for navigating the class hierarchy if you want to be able to see both superclasses and subclasses at the same time.

Click on the Hierarchy tab to browse classes with the hierarchy view. The hierarchy view shown in Figure 7.6 appears.

Figure 7.6 Viewing superclass and subclass information in the Class Browser



The areas available in the hierarchy view are described below.

7.3.1 Class box

As with other views in the Class Browser, the name of the class being browsed is given here. See Section 7.2.1 on page 80 for more details.

7.3.2 Superclasses list

This list displays the immediate superclasses of the current class. Double-clicking on any class makes it the current class.

Selecting a class in this list displays its description in the Description list.

7.3.3 Subclasses list

This list displays the immediate subclasses of the current class. Double-clicking on any class makes it the current class.

Selecting a class in this list displays its description in the Description list.

7.3.4 Description list

This list displays a description of the first class selected in either the Superclasses or Subclasses lists, or the current class if there is no selection in either of these lists. The following information is printed:

Package	The name of the package that the selected class is defined in.
Name	The name of the selected class.
Metaclass	The class of the selected class. The metaclass is the class of Lisp object that the current class belongs to.
Accessibility	The accessibility of the selected class—whether the symbol is external or internal, as returned by **`find-symbol`**.

To operate on any of the items displayed in this area, select them and choose a command from the **Description** menu. This menu contains the standard actions commands described in Section 3.7 on page 35. You can operate on more than one item at once by making a multiple selection in this area.

7.3.5 Performing operations on the selected classes or the current class

You can use the **Classes** menu to perform operations on any number of items selected in either the Subclasses area or the Superclasses area. If no items are selected, then the

current class is operated on by the commands in this submenu. The standard actions commands described in Section 3.7 on page 35 are available in this submenu.

Choose **Classes > Browse Metaclass** to select, and describe in the normal way, the class of the selected classes, or the current class.

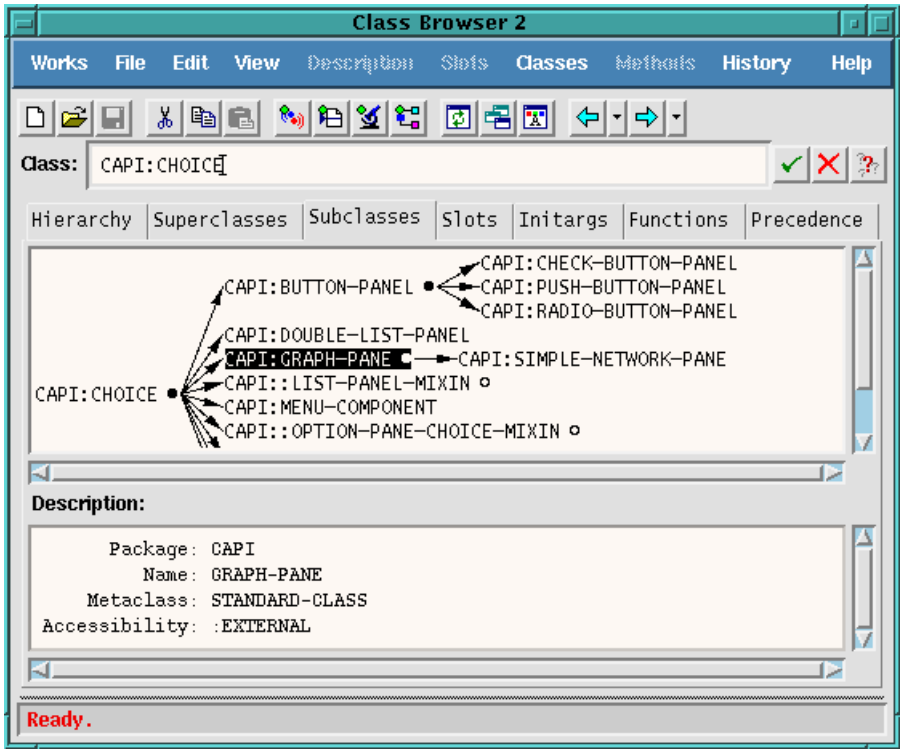
Note: If more than one item is selected, and the command chosen from the **Classes** menu invokes a tool which can only display one item at a time, then the extra items are added to the **History > Items** submenu of the tool, so that you can easily display them.

7.4 Examining classes graphically

As already mentioned, you can view class relationships graphically using either the superclasses or subclasses views. This gives an immediate impression of the class hierarchy, but contains no details about information such as slots, readers and writers.

Click on the Subclasses tab to browse subclasses in a graph, and click on the Superclasses tab to view superclasses in a graph. Except for the type of information shown, these two views are visually identical. The subclasses view is shown in Figure 7.7.

Figure 7.7 Viewing subclasses graphically in the Class Browser



The areas available in the subclasses and superclasses views are described below.

7.4.1 Class box

As with other views in the Class Browser, the name of the class being browsed is shown here. See Section 7.2.1 on page 80 for details.

7.4.2 Subclasses and superclasses graphs

The main area of these views is a graph showing either the subclasses or the superclasses of the current class, depending on the view you have chosen. The generic facil-

ities available to all graph views throughout the environment are available here: see Chapter 5, “Manipulating Graphs” for details.

Selecting a node in this displays a description of the class it represents in the Description list.

7.4.3 Description list

This list displays a description of the first class selected in the graph. This gives the same information as the Description list in the hierarchy and precedence views. See Section 7.3.4 for details.

7.4.4 Performing operations on the selected classes or the current class

You can operate on the selected node in the graph using the commands in the **Classes** menu. If no node is selected, then the current class is operated on by the commands in this menu. The standard actions commands described in Section 3.7 on page 35 are available in this menu.

Choose **Classes > Browse Metaclass** to select, and describe in the normal way, the class of the selected classes, or the current class.

7.4.5 An example

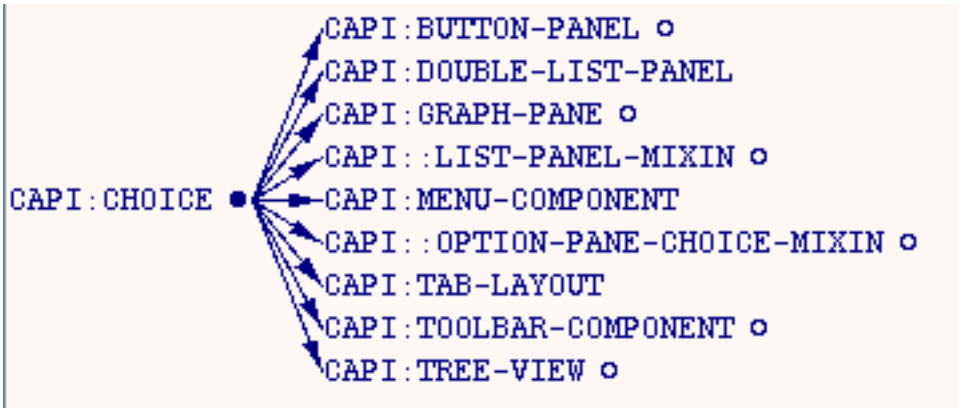
1. Examine the class **`capi:choice`** by typing **`capi:choice`** into the Class area of the Class Browser and pressing **`Return`** or clicking on ✓.

   The class is described in the current view.

2. Click on the Subclasses tab in the Class Browser.

The relationships between **capi:choice** and its subclasses are shown in a graph, as in Figure 7.8.

Figure 7.8 Relationship between **capi:choice** class and its subclasses



By default, the subclasses of the current class are shown in the graph. To expand a non-leaf node in the graph, click on the circle to its right.

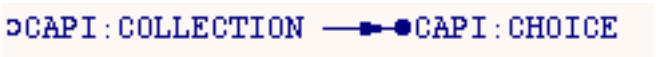
3. Expand the **CAPI:BUTTON-PANEL** node to see the subclasses of this class.

   The classes of button panel object available are displayed in the graph, including the push button panel class that you saw in the examples in Section 7.1 on page 72.

4. To graph the superclasses, click the Superclasses tab.

   The relationships between **capi:choice** and its superclasses are shown in a graph, as in Figure 7.9.

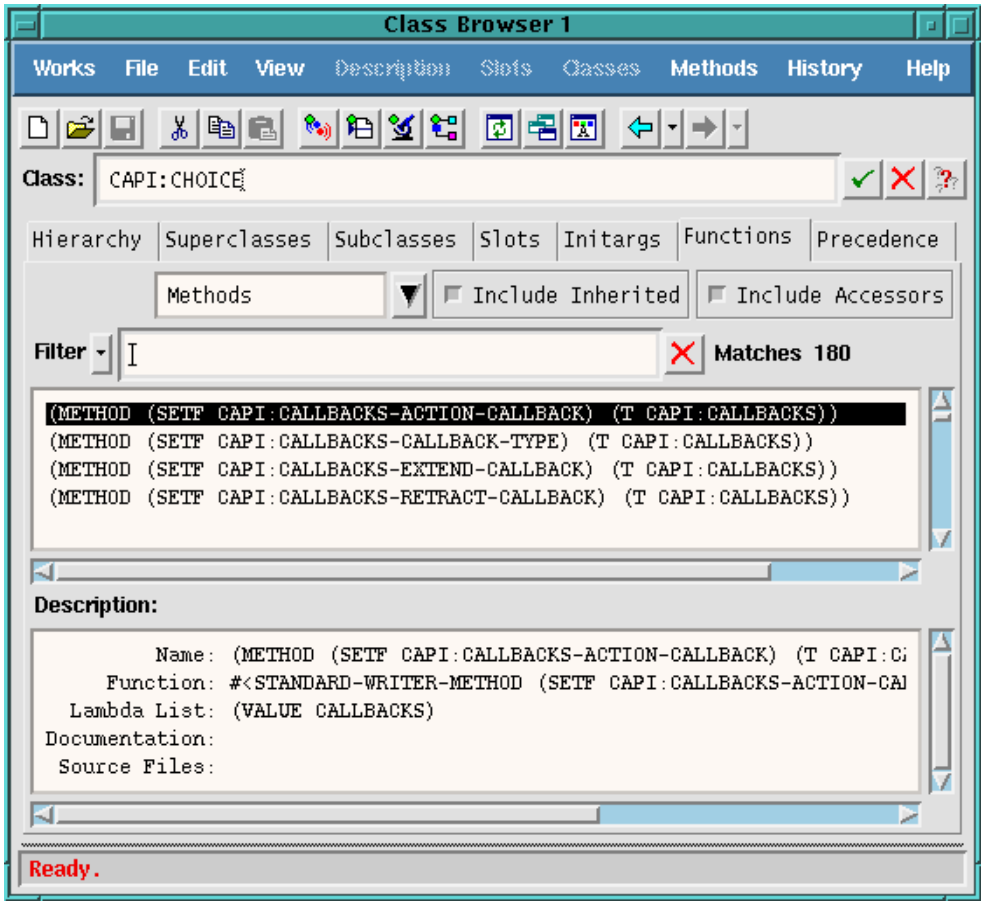
   Figure 7.9 Relationship between **capi:choice** class and its superclasses



7.5 Examining generic functions and methods

Click the Functions tab to examine information about the generic functions and methods defined on the current class. The functions view shown in Figure 7.10 appears.

Figure 7.10 Displaying function information in the Class Browser



This view can be especially useful when used in conjunction with the Generic Function Browser. The areas available are described below.

7.5.1 Class box

As with other views in the Class Browser, the name of the class being browsed is given here. See Section 7.2.1 on page 80 for more details.

7.5.2 Filter box

The Filter box lets you restrict the information displayed in the list of functions or methods. See “Filtering information” on page 42 for a description of how to use the Filter box in any tool, and Section 7.1.3 on page 75 for an example of how to use it in the Class Browser.

7.5.3 List of functions or methods

This lists either the generic functions with applicable methods for the current class, or the applicable methods for the current class. Items selected in this list can be operated on via the **Methods** menu, as described in Section 7.5.6 on page 90. Double-clicking on a function or method displays its source code definition in the Editor, if possible.

Select **Methods** or **Generic Functions** from the drop-down list box to choose which type of information to list.

If **Include Inherited** is checked, generic functions or methods inherited from the super-classes of the current class are displayed.

If **Include Accessors** is checked, accessor methods/functions are displayed. When **Include Accessors** is not checked, methods/functions defined by the `:readers`, `:writers` and `:accessors` slot options in `defclass` are omitted from the display.

You can alter the default settings of these options in the **Slots/Functions** tab of the Class Browser Preferences dialog. Choose **Works > Tools > Preferences...** to display this dialog.

7.5.4 Description list

The list at the bottom of the tool gives a description of the function or method selected in the main list. The following information is shown:

Name	The name of the selected generic function or method.

Function	The function which the selected function or method relates to.
Lambda List	The lambda list of the selected generic function or method.
Documentation	The Common Lisp documentation for the selected function or method, if any exists.
Source Files	The source files for the selected generic function or method.

To operate on any of the items displayed in this area, select them and choose a command from the **Description** menu. This submenu contains the standard actions commands described in Section 3.7 on page 35. You can operate on more than one item at once by making a multiple selection in this area.

7.5.5 Performing operations on the current class

You can operate on the current class using the commands in the **Classes** menu. The standard action commands described in Section 3.7 on page 35 are available from this submenu.

Choose **Classes > Browse Metaclass** to select and describe the class of the current class.

7.5.6 Operations specific to the current function or method

In addition to the commands described above, the following commands are available when using the functions view.

The standard action commands described in Section 3.7 on page 35 are available from the **Methods** menu.

Choose **Methods > Undefine...** to remove the selected functions or methods from the Lispworks image. You are prompted before the functions or methods are removed.

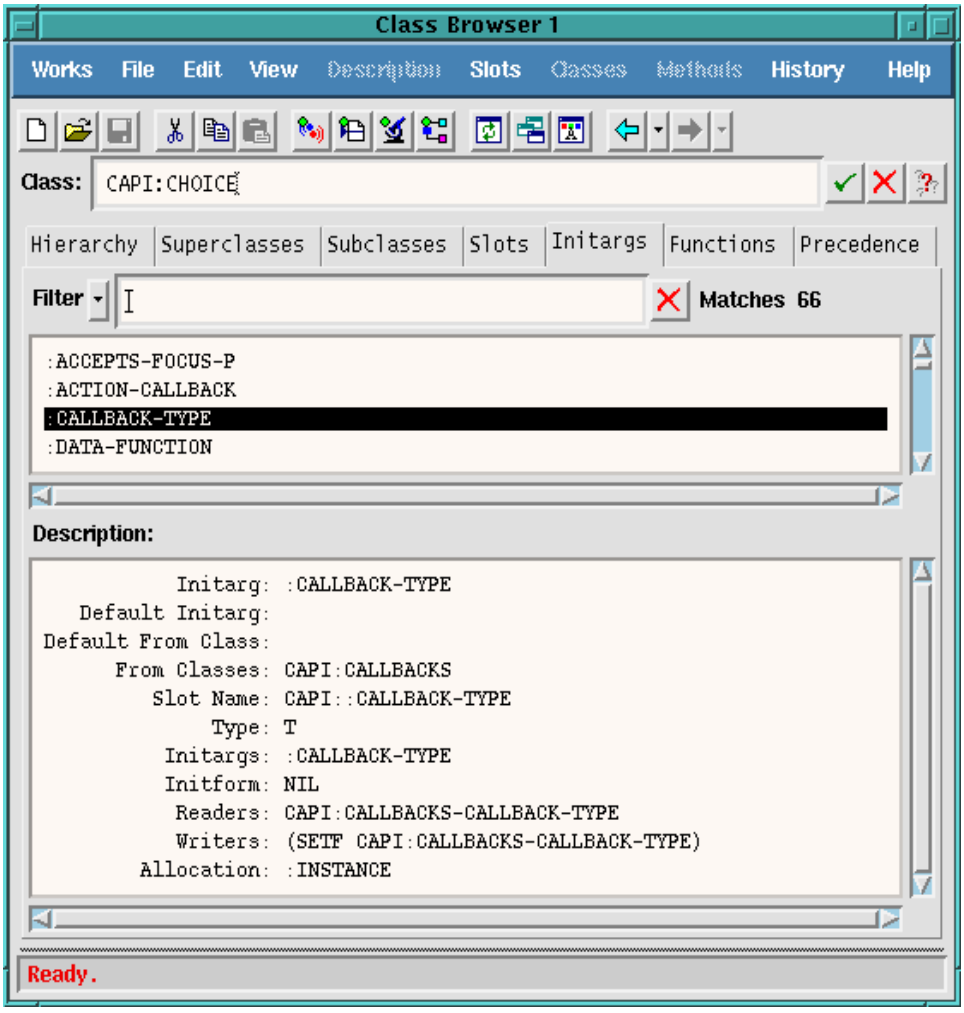
Warning: Do not remove system functions and methods, such as those defined for CAPI classes used as examples in this chapter.

Choose **Methods > Trace** to display the **Trace** submenu available from several tools. This submenu lets you trace the selected methods or generic functions. A full description of the commands in this submenu is given in Section 3.9 on page 40.

7.6 Examining initargs

Click the Initargs tabs to examine information about the initargs of the current class. The initargs view shown in Figure 7.11 appears.

Figure 7.11 Displaying initarg information in the Class Browser



The areas available are described below.

7.6.1 Class box

This area gives the name of the class being browsed. See Section 7.2.1 on page 80 for details.

7.6.2 Filter box

The Filter box lets you restrict the information displayed in the initargs list. See “Filtering information” on page 42 for a description of how to use the Filter box in any tool, and Section 7.1.3 on page 75 for an example of how to use it in the Class Browser.

7.6.3 List of initargs

This lists the slots in the current class for which initargs have been defined. Selecting an item in this list displays information in the Description list. Any items selected can also be operated on via the **Slots** menu.

7.6.4 Description list

This area gives a description of the initarg selected in the Initargs area. The following items of information are displayed:

Initarg	The name of the selected initarg.
Default Initarg	The default value for the selected initarg, if defined with `:default-initargs`.
Default From Class	The class providing the default for the initarg.
From Classes	The class from which the selected initarg is inherited.
Slot Name	The name of the slot to which this initarg relates.
Type	The type of the selected initarg.
Initargs	All initargs applicable to the same slot.

Initform	The initform for the slot to which this initarg relates.
Readers	The readers for the slot to which this initarg relates.
Writers	The writers for the slot to which this initarg relates.
Allocation	The allocation for slot to which this initarg relates. See CLOS in the ANSI Common Lisp specification for details.

Items selected in this list can be operated on via the **Description** menu.

7.6.5 Performing operations on the current class

You can operate on the current class using commands in the **Classes** menu. The standard action commands described in Section 3.7 on page 35 are available in this submenu.

Choose **Classes > Browse Metaclass** to select, and describe in the normal way, the class of the current class.

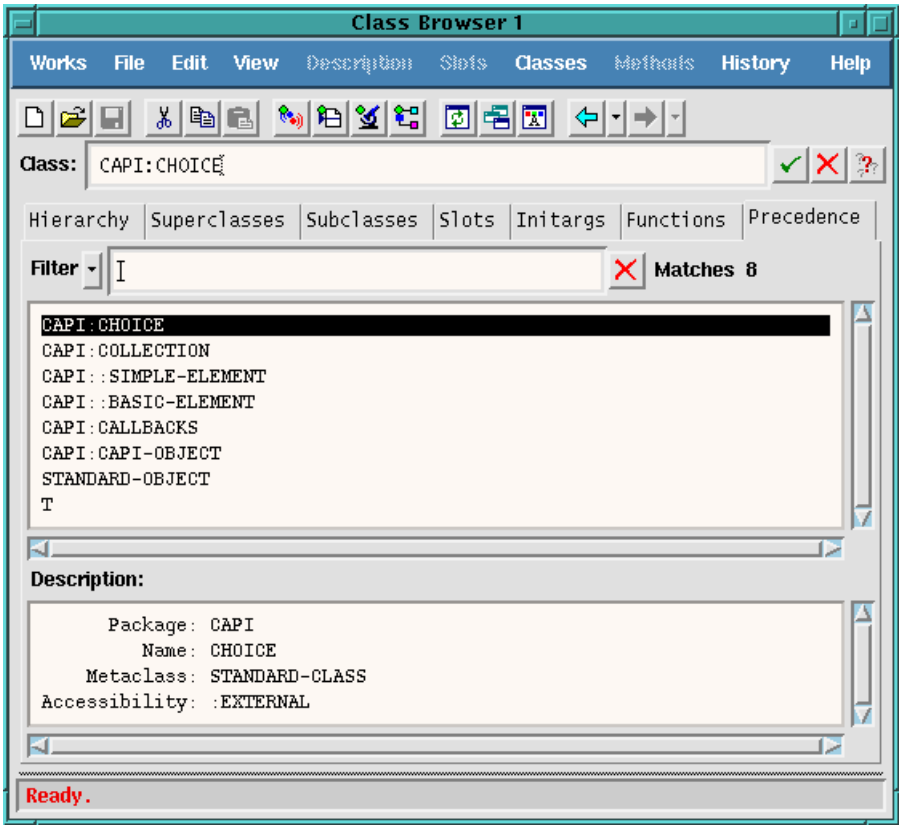
7.7 Examining class precedences

Click the Precedence tag to examine information about the precedence list of the current class. The precedence view shown in Figure 7.12 appears.

The precedence list is used to generate the method combinations for a class, and thus can be used to tell you which method applies in a given case.

See Chapter 14, “The Generic Function Browser” for details on examining information about methods.

Figure 7.12 Displaying precedence information in the Class Browser



The areas available are described below.

7.7.1 Class box

As with all other views in the Class Browser, the current class is printed in this area. See Section 7.2.1 on page 80 for full details of its use.

7.7.2 Filter box

The Filter box lets you restrict the information displayed in the list of precedences. See “Filtering information” on page 42 for a description of how to use the Filter box in any tool, and Section 7.1.3 on page 75 for an example of how to use it in the Class Browser.

7.7.3 List of precedences

This list is the class precedence list of the current class. Precedences are listed highest first. Double-clicking on an item in this list describes that class in the Class Browser.

7.7.4 Description list

This gives the same class description available in the superclasses, subclasses, and hierarchy views. See Section 7.3.4 on page 83 for details.

7.7.5 Performing operations on the selected classes or the current class

You can operate on any number of selected items in the list of precedences using the commands in the **Classes** menu. If no items are selected, then the current class is operated on by the commands in this submenu. The standard actions commands described in Section 3.7 on page 35 are available in this submenu.

Choose **Classes > Browse Metaclass** to select, and describe in the normal way, the class of the selected classes, or the current class.

Note: If more than one item is selected, and the command chosen from the **Classes** menu invokes a tool which can only display one item at a time, then the extra items are added to the **History > Items** submenu of the tool, so that you can easily display them.

8

The Object Clipboard

The Object Clipboard is a utility that allows you to keep track of multiple Lisp objects as you examine and manipulate them with the Common LispWorks tools.

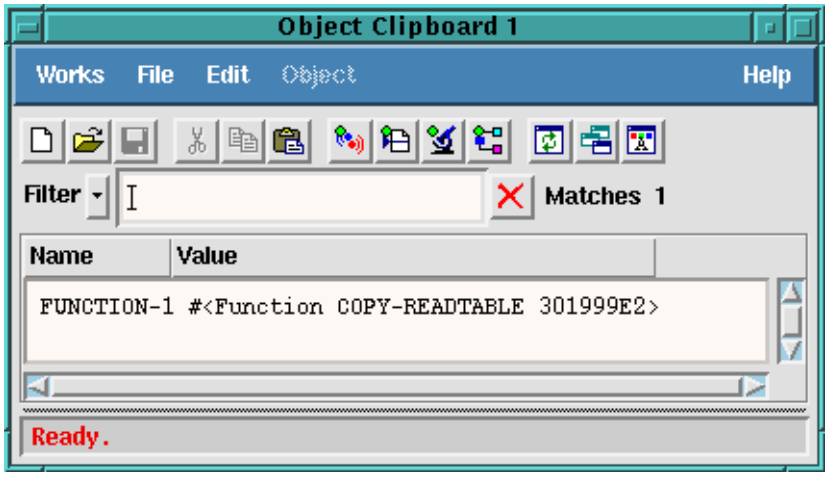
Recall that a Lisp object which is viewed in some tool can be temporarily stored and then pasted into another tool. See the descriptions of the **Copy**, **Cut** and **Paste** commands in “Using the clipboard” on page 28 and “Operations available” on page 36.

The Object Clipboard, and its associated **Clip** command provides a more powerful mechanism whereby multiple Lisp objects can be stored ("clipped") and later retrieved.

Note: the **Clip** command retains a pointer to the clipped object even if you do not have an Object Clipboard tool visible. When you create the tool, the clipped objects are visible in it

To create an Object Clipboard tool, choose **Works > Tools > Object Clipboard** or click in the Podium.

Figure 8.1 The Object Clipboard



The Object Clipboard creates a name for the clipped object based on its type, and shows the object itself in the Value column.

8.1 Placing objects on the Object Clipboard

You can place an object on the Object Clipboard by using the menu command **Clip**, available in most tools as described below..

8.1.1 The Listener

To place the current object of a Listener on the Object Clipboard (that is, the value of the variable `cl:*`), choose **Values > Clip** in the Listener.

If your Listener is in the debugger, you can clip the condition object by **Debug > Condition > Clip**.

8.1.2 The Class Browser

To place a class from the Class Browser on the Object Clipboard, select the class name in the Hierachy, Superclasses, Subclasses or Precedence tab, and choose **Classes > Clip**.

To place a slot definition object from the Class Browser on the Object Clipboard, select the slot name in the Slots tab, and choose **Slots > Clip**.

To place a method or generic function object from the Class Browser on the Object Clipboard, select it in the Functions tab, and choose **Methods > Clip**.

8.1.3 The Inspector

To place the currently inspected object in the Inspector on the Object Clipboard, choose **Object > Clip**.

To place the value in a slot of the currently inspected object, select the slot in the Inspector, and choose **Slots > Clip**.

8.1.4 The Function Call Browser

To place the current function on the Object Clipboard, choose **Function > Clip**. If you have selected a function name in the Function Call Browser, that function is clipped instead.

8.1.5 The Generic Function Browser

To place a method from the Generic Function Browser on the Object Clipboard, select the method and choose **Methods > Clip**. For the generic function object itself, choose **Function > Clip**.

8.1.6 The Debugger

To place the condition object from the Debugger tool on the Object Clipboard, choose **Condition > Clip**.

To place the value of a variable in the Debugger's backtrace area on the Object Clipboard, select the variable and choose **Variables > Clip**.

8.1.7 The Stepper

To place the value of a variable in the Stepper's Backtrace tab onto the Object Clipboard, select the variable and choose **Variables > Clip**.

8.1.8 The System Browser

To place the system object from the System Browser onto the Object Clipboard, choose **Systems > Clip**.

8.1.9 General clipping

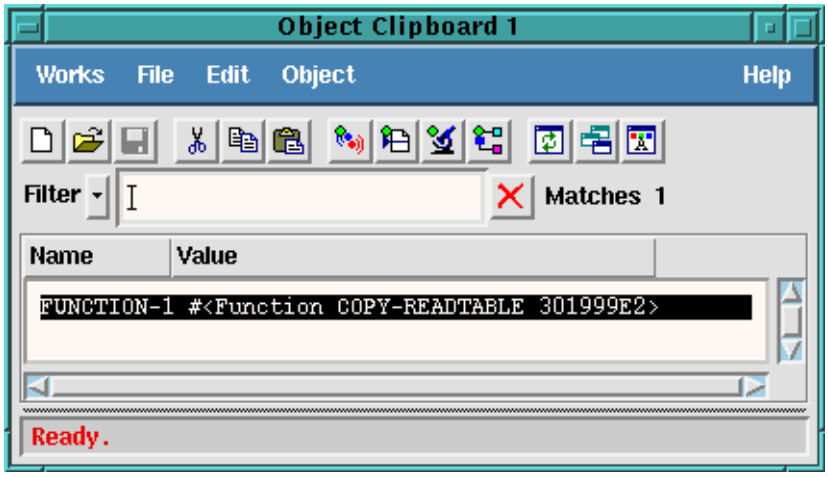
To place any CAPI top level window itself on the Object Clipboard, choose **Works > Interface > Clip**.

To place data from a Description panel, such as in the Class Browser or in the Tree tab of the Compilation Conditions Browser, select the desired parts of the Description and choose **Description > Clip**.

8.2 Browsing clipped objects

For each object in the Object Clipboard, you can can browse it in various tools as described below. First, select the object you want to browse and note that the **Object** menu is enabled:

Figure 8.2 An object selected in the Object Clipboard



8.2.1 The Inspector

To inspect any object that is on the Object Clipboard, select it and choose **Object > Inspect**.

8.2.2 The Class Browser

To browse the class of any object that is on the Object Clipboard, select it and choose **Object > Class**.

8.2.3 The Listener

To paste an object from the Object Clipboard into the Listener, choose **Object > Listen**.

8.2.4 General browsing

To browse an object that is on the Object Clipboard, select it and choose the Browse command from the **Object** menu. For example, if the object is a generic function, the menu command is **Object > Browse - Generic Function**.

8.2.5 Pasting of clipped objects

This is another way to view a clipped object in another tool.

Paste an object from the Object Clipboard into another tool by:

1. Select the object in the Object Clipboard window
2. Choose **Edit > Copy**.
3. Make the other tool window active.
4. Choose **Edit > Paste**.

8.3 Removing objects

To remove an object from the Object Clipboard, select it and choose **Edit > Cut**.

To empty the Object Clipboard, first remove any filter. Then choose **Edit > Select All** followed by **Edit > Cut**.

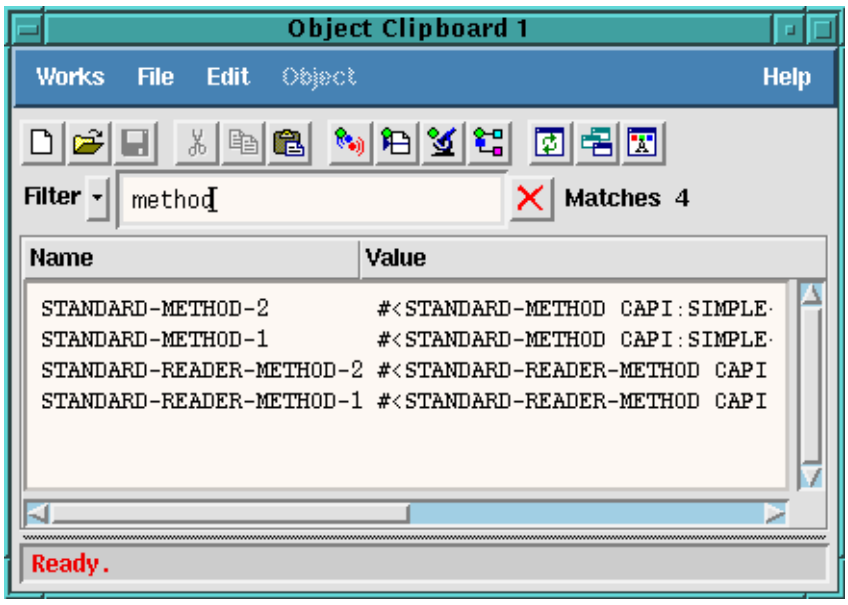
Note: if you close the Object Clipboard window, the objects are not removed. They are preserved and displayed in a subsequently created Object Clipboard windows.

8.4 Filtering

You can use the Filter box of the Object Clipboard in the standard way to reduce the number of clipped objects displayed.

For example to see only the method objects in the Object Clipboard, enter "method" in the Filter box.

Figure 8.3 Use of the Filter box in the Object Clipboard



For more information about filtering, see “Filtering information” on page 42.

8.5 Using the Object Clipboard with a Listener

Here we place several objects on the Object Clipboard. Then we link the Object Clipboard with a Listener tool, giving a convenient way to manipulate these objects in turn.

In the Listener:

1. Enter

```
(capi:contain
 (make-instance 'capi:display-pane
                :text "Display Pane"
                :background :green))
```

   A green display pane is displayed.

2. Ensure that the Listener window is active, so that the **Values** menu is enabled. Choose **Values > Clip** to place the display pane on the Object Clipboard.

3. Enter

```
(capi:contain
 (make-instance 'capi:editor-pane
                :text "Editor Pane"
                :background :yellow))
```

   A yellow editor pane is displayed.

4. Return to the Listener and choose **Values > Clip** to place the editor pane on the Object Clipboard.

5. Enter

```
(capi:contain
 (make-instance 'capi:graph-pane))
```

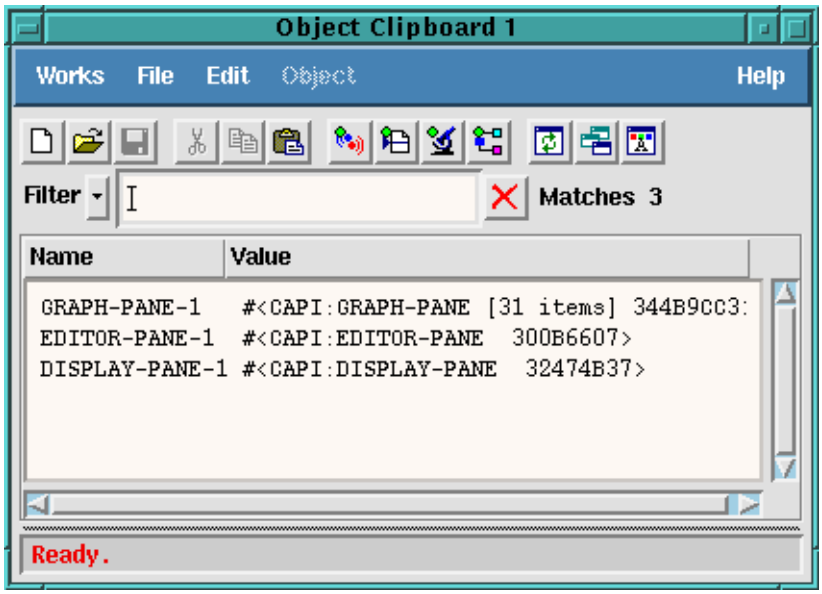
   A graph pane is displayed.

6. Return to the Listener and choose **Values > Clip** to place the graph pane on the Object Clipboard.

Now choose **Tools > Object Clipboard**or click in the Podium. Notice that this creates an Object Clipboard tool if you do not already have one. The Object Clipboard

shows the objects you just clipped, and the most recently clipped object appears at the top. It should look like Figure 8.4.

Figure 8.4 CAPI panes in the Object Clipboard



In the Listener choose **Edit > Link from** and select the Object Clipboard in the sub-menu. Now, whenever you select an object in the Object Clipboard, it is also pasted into the Listener - that is, it becomes the value of `*`. We use this link to manipulate the CAPI pane objects in the Listener.

1. In the Object Clipboard select DISPLAY-PANE-1. This raises the linked Listener window and pastes the display pane object.
2. Enter in the Listener:

   ```
   (capi:apply-in-pane-process
    * #'(setf capi:simple-pane-background) :red *)
   ```

   The display pane background becomes red.
3. In the Object Clipboard select EDITOR-PANE-1. This raises the linked Listener window and pastes the editor pane object.

4. In the Listener choose **History > Previous** or use **Esc P**, and press **Return**, to enter the same command again

```
(capi:apply-in-pane-process
 * #'(setf capi:simple-pane-background) :red *)
```

   The editor pane background also becomes red.

5. In the Object Clipboard select GRAPH-PANE-1. This raises the linked Listener window and pastes the graph pane object.

6. Enter in the Listener:

```
(capi:apply-in-pane-process
 * #'(setf capi:graph-pane-roots) '( 2 3) *)
```

   The graph pane is altered.

Notice how linking the Listener with the Object Clipboard allows you to manipulate the clipped objects in turn via the value of *****.

9

The Compilation Conditions Browser

9.1 Introduction

The Compilation Conditions Browser gives you an interface to the warning and error conditions you are likely to encounter when compiling your source code. It allows you to see the relationship between different errors or warnings encountered during compilation, and gives you immediate access to the source code which produced them.

You can use it to view the conditions signaled during compilation of files from any part of the environment: whether you are compiling files using the System Browser or the Editor, any ensuing conditions can be displayed in the Compilation Conditions Browser. The Compilation Conditions Browser requires the source code to come from a file.

The Compilation Conditions Browser has two views.

- The Conditions view, which shows the conditions grouped by file name.
- The Output view, which can be used to display the output messages in the environment.

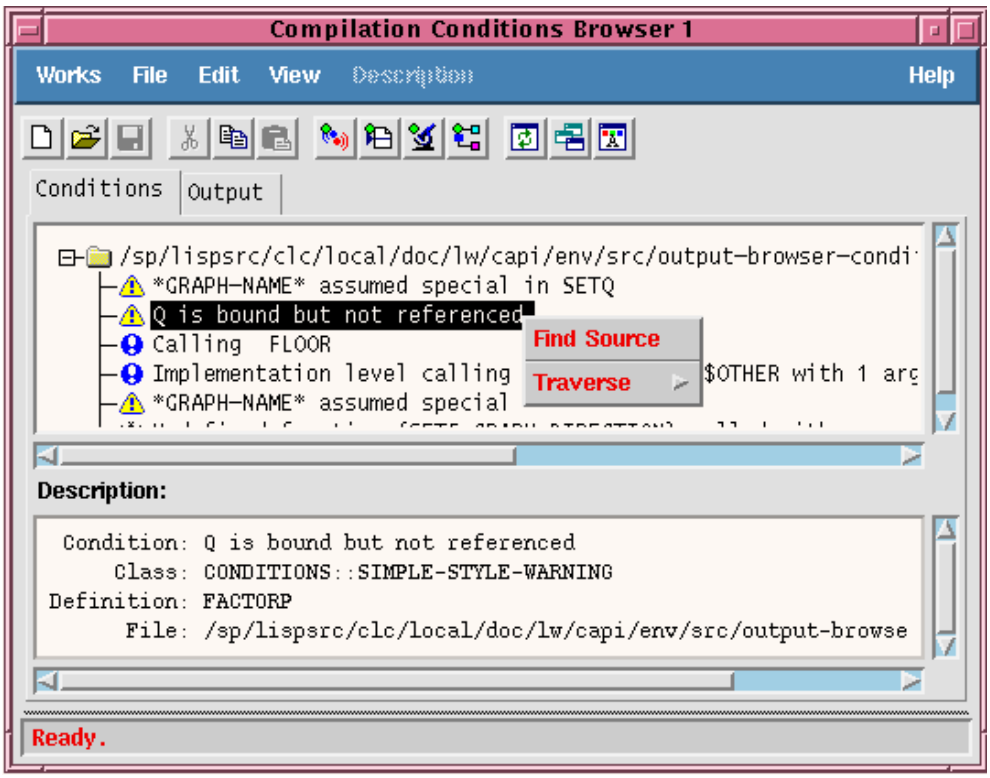
To create a Compilation Conditions Browser, you can choose **Works > Tools > Compilation Conditions Browser** or click in the Podium.

A more common way to create a Compilation Conditions Browser is to press **Return** when the Output tab (of any tool) reports compilation conditions. See “Compiling in memory” on page 180 for details.

9.2 Examining error conditions

The Conditions view is the default view in the Compilation Conditions Browser, and is therefore visible when the browser is first invoked. The Compilation Conditions Browser appears as shown in Figure 9.1.

Figure 9.1 The Compilation Conditions Browser



There are two tabs. These show the same information, in different ways:

- Conditions - default view that shows all conditions in a tree representation, grouped by filename. Each item in the tree can be expanded to show the conditions that were generated during compilation of that file. Selecting a compila-

tion message in the tree view causes the data for the selected message to be shown in the Description area. Double-clicking on an item (or using **Find Source** on the context menu, as illustrated above) shows the source code of the condition in an Editor, highlighting the nearest subform to where the condition occurred. After doing this, **Ctrl+X `** (backquote) can be used to find the source of the next condition shown in the browser.

- Output - shows the raw compilation output. You will see this same output in the tool that performed the compilation.

The description area in the Conditions view of the Compilation Conditions Browser shows a description of any item selected in the conditions area. The description contains details of the selection condition. The following information is shown:

Condition	The error condition for the selected item in the message area.
Class	The class of the selected condition.
Definition	The name of the form in which the condition was signaled.
File	The name of the file that contains the Lisp source code that caused the selected condition.

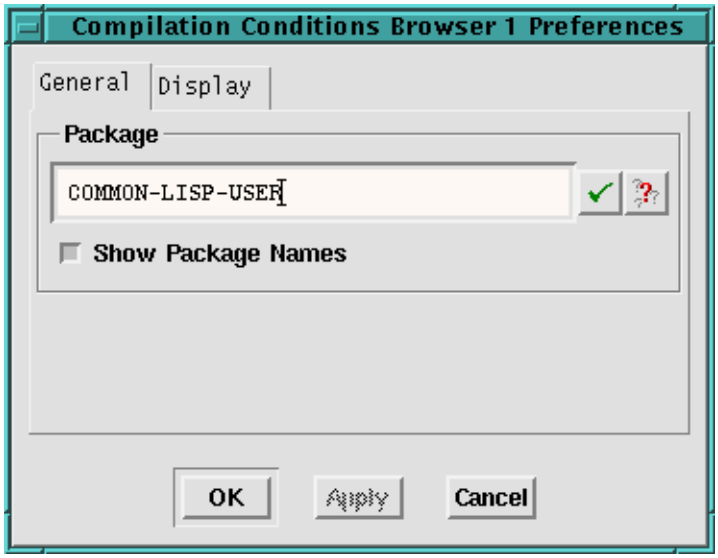
Items selected in this area may be examined using the **Description** menu which allows a variety of LispWorks tools to be invoked on the selected item in the description area.

9.3 Configuring the display

The manner in which the Compilation Conditions Browser displays information can be customized by selecting **Works > Tools > Preferences...** in the browser, to invoke

the Compilation Conditions Browser preference dialog. When first invoked, it appears as shown in Figure 9.2, with the general view active.

Figure 9.2 The Compilation Conditions Browser preferences dialog



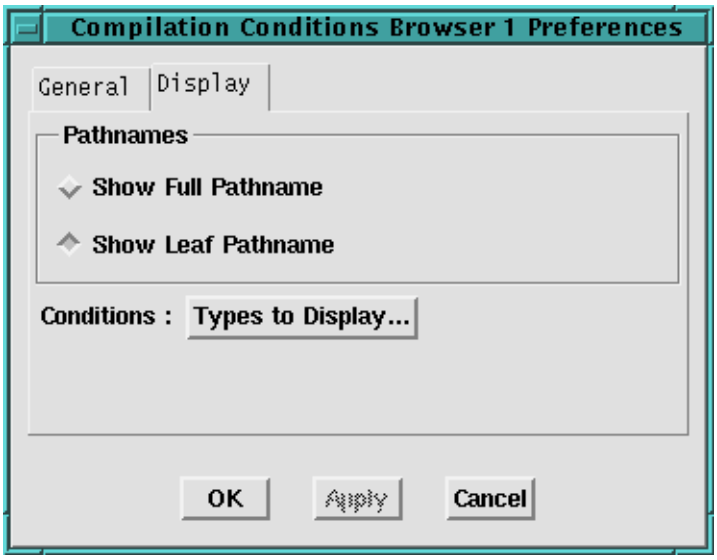
9.3.1 Package names preferences

In the general view select **Show Package Names** to toggle display of packages in all references to symbols. Use the package selection box to select which package to show. The default package is **CL-USER**. Setting the right package and turning off package names can greatly simplify a complicated list.

9.3.2 Pathname preferences

In the display view, the preferences dialog appears as in Figure 9.3.

Figure 9.3 The display view of the preferences dialog



The major part of this dialog is taken up by the pathnames selection area, which has two radio buttons.

Check **Show Full Pathname** to show the full pathname of all files displayed. This is the default setting.

Check **Show Leaf Pathname** to show just the filename of all files displayed, and omit the full pathname.

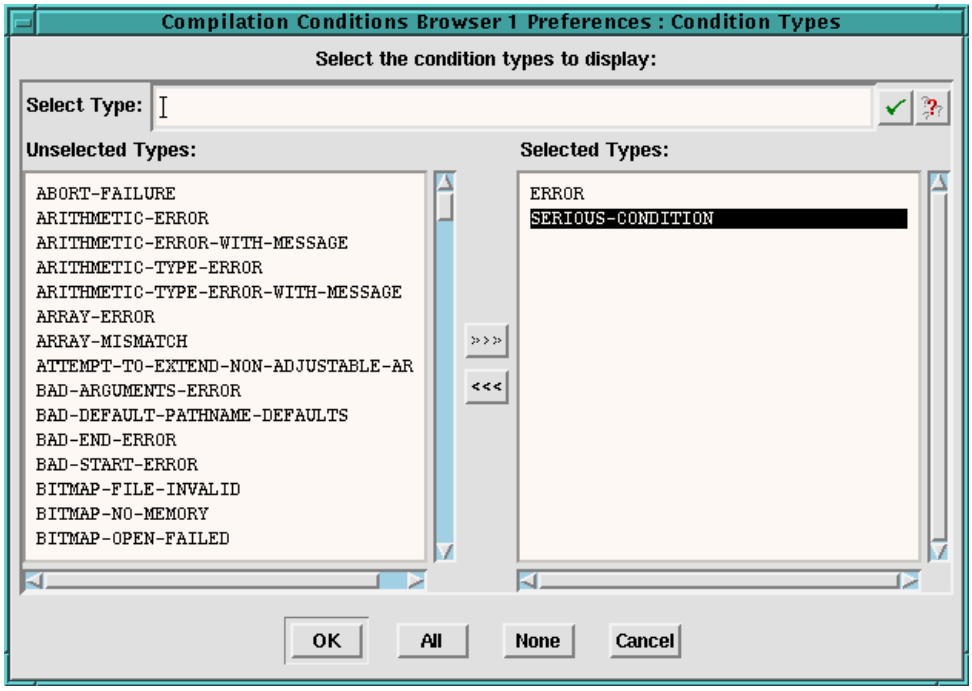
9.3.3 Condition type preferences

Clicking on **Types to Display...** in the display view of the preferences dialog calls up the condition types dialog as shown in Figure 9.4, page 112. It consists of a **Select Type:** box and two lists.

- The **Selected Types** list shows the condition types that are displayed in the browser.

- The **Unselected Types** list shows the condition types that are not displayed in the browser.

Figure 9.4 The Condition Types dialog



By default, the **condition** type is selected as it includes all subclasses.

To remove a condition type from the **Selected Types** list, select it and click on **<<<**. It is transferred into the **Unselected Types** list.

To add a condition type back into the **Selected Types** list, either:

- select it in the **Unselected Types** list and click on **>>>**, or
- Enter its name in the **Select Type:** area and press **Return**.

If you use the second of these methods, you can press which allows you to select from a list of all condition type names which begin with the partial input you have entered. See “Completion” on page 46 for detailed instructions.

To display all condition types, click on **All**.

To display no condition types, click on **None**. If you only want to display a few condition types, click on this button and then transfer the error types you do want to see into the **Selected Types** list using one of the methods described above.

When you have finished choosing the condition types, click on **OK**. The dialog is dismissed, and the Compilation Conditions Browser is updated to display the condition types you have requested.

Click on **Cancel** to cancel the dialog. The dialog is dismissed, and no changes are made to the display.

9.4 Access to other tools

The Compilation Conditions Browser is integrated with the other tools allowing intuitive interaction.

You can easily find the source the generated a condition, as described in “Examining error conditions” on page 108.

Items selected in the Description area may be examined using the **Description** menu. See “Operations available” on page 36 for more information on the operations available from this menu. Additionally, double-clicking on part of the description displays it in an Inspector or Class Browser, as appropriate.

10

The Debugger Tool

When developing source code, mistakes may prevent your programs from working properly, or even at all. Sometimes you can see what is causing a bug in a program immediately, and correcting it is trivial. For example, you might make a spelling mistake while typing, which you may instantly notice and correct.

More often, however, you need to spend time studying the program and the errors it signalled before you can debug it. This is especially likely when you are developing large or complex programs.

A Debugger tool is provided to make this process easier. This tool is a graphical front-end to the command line debugger which is supplied with your Lisp image. In order to get the best use from the Debugger tool, it is helpful if you are familiar with the command line debugger supplied. See the *LispWorks User Guide* for a description of the command line debugger.

The Debugger tool can be used to inspect programs which behave in unexpected ways, or which contain Common Lisp forms which are syntactically incorrect.

There are two ways that you can invoke the Debugger tool:

- If you evaluate code that signals an error in a Listener, the command line debugger is entered automatically. At this point, choose **Debug > Start GUI Debugger** or press the button in the Listener toolbar to invoke the Debugger tool.

- If you run code that signals an error from another source (for example, as a result of running a windowed application, or compiling code in a file of source code), a notifier window appears. Click on the **Debug** button in this notifier to invoke the Debugger tool.

Here is a short example introducing the Debugger tool:

1. Define the following function in the Listener.

   ```
   (defun thing (number)
     (/ number 0))
   ```

   This function which attempts to divide a number given as an argument by zero.

2. Now call this function as follows:

   ```
   (thing 12)
   ```

   The call to **thing** invokes the command line debugger.

3. Choose **Debug > Start GUI Debugger** or press the button to invoke the Debugger tool. Notice that the window title contains the name of the process being debugged.

4. For now, click the **Abort** button in the Debugger toolbar to return to the top level loop in the Listener.

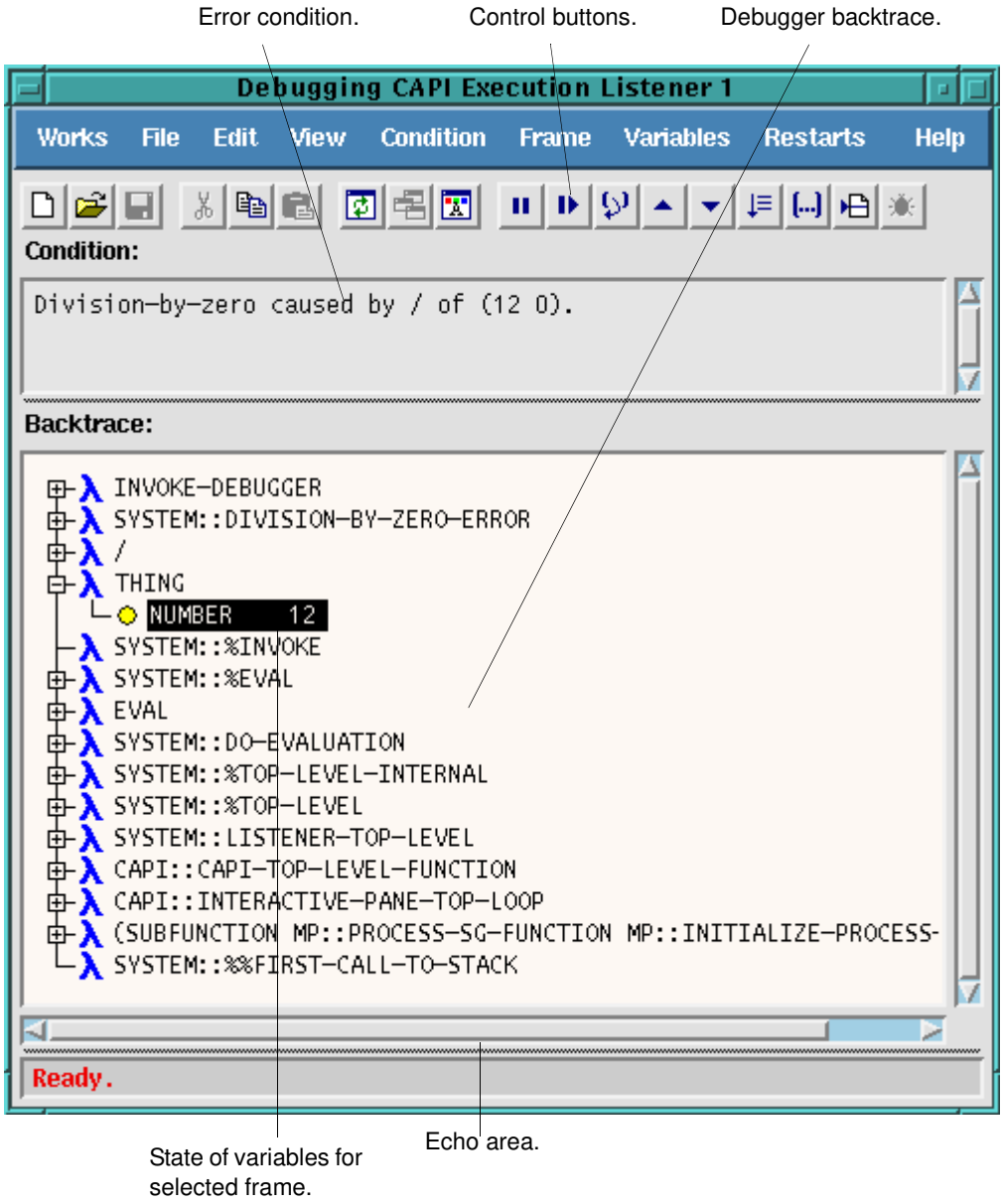
The command line debugger can be entered by signaling an error in interpretation or execution of a Common Lisp form. For each error signaled, a further level of the debugger is entered. Thus, if, while in the debugger, you execute code which signals an error, a lower level of the debugger is entered. The number in the debugger prompt is incremented to reflect this.

Note that you can also invoke the command line debugger by tracing a function and forcing a break on entry to or exit from that function. See the tutorial chapter (Section 2.3) for the example code used in Figure 2.4 and Figure 10.1.

10.1 Description of the Debugger

By default the debugger tool appears as shown in Figure 10.1 below.

Figure 10.1 Debugger tool



The debugger tool has two areas, and a toolbar. These are described below. If you invoke the debugger tool by clicking **Debug** in a notifier window, the tool also con-

tains a listener pane. This provides you with a useful way of evaluating Common Lisp forms interactively in the context of the error.

10.1.1 Condition box

This area displays the error condition which caused entry to the debugger. You cannot edit the text in this box.

The error condition can be operated on by commands in the **Condition** menu. See "Performing operations on the error condition" on page 127 for details.

10.1.2 Backtrace area

The backtrace area displays the function calls on the execution stack. Each tree root or list item in the backtrace area represents a stack frame associated with a function call.

Double-clicking on any stack frame finds and displays the source code definition for that function in the Editor, if this is known. Any frame selected in this area can be operated on using the commands in the **Frame** menu, which is also available as the context menu. See "Performing operations on stack frames" on page 127 for details.

The backtrace is displayed either in a tree or a list, with the behaviors described below.

You can choose which type of display it uses by the **Frames and Arguments** preference, described in "Configuring the debugger tool" on page 129.

10.1.2.1 Frames and Variables in a tree

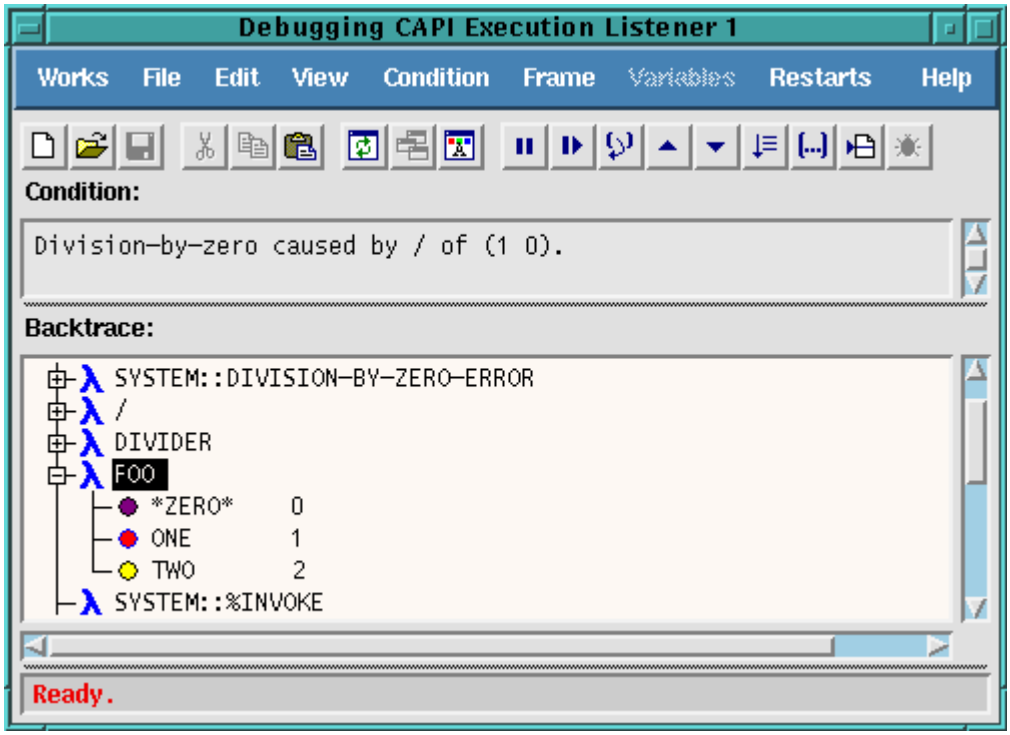
When the **Frames and Arguments** preference has the value **Tree-view**, the Debugger appears as shown in Figure 10.2 below.

Each expandable root node in the **Backtrace:** tree represents a stack frame associated with a function call. You can operate on the frame as described in "Backtrace area" on page 119.

Expanding a stack frame node displays any variables associated with that function call. You can double click on any variable to inspect it using the Inspector tool. Any

items selected in this area can be operated on using the commands in the **Variables** menu: see “Performing operations on frame variables” on page 128 for details.

Figure 10.2 Variables in the Debugger tree view



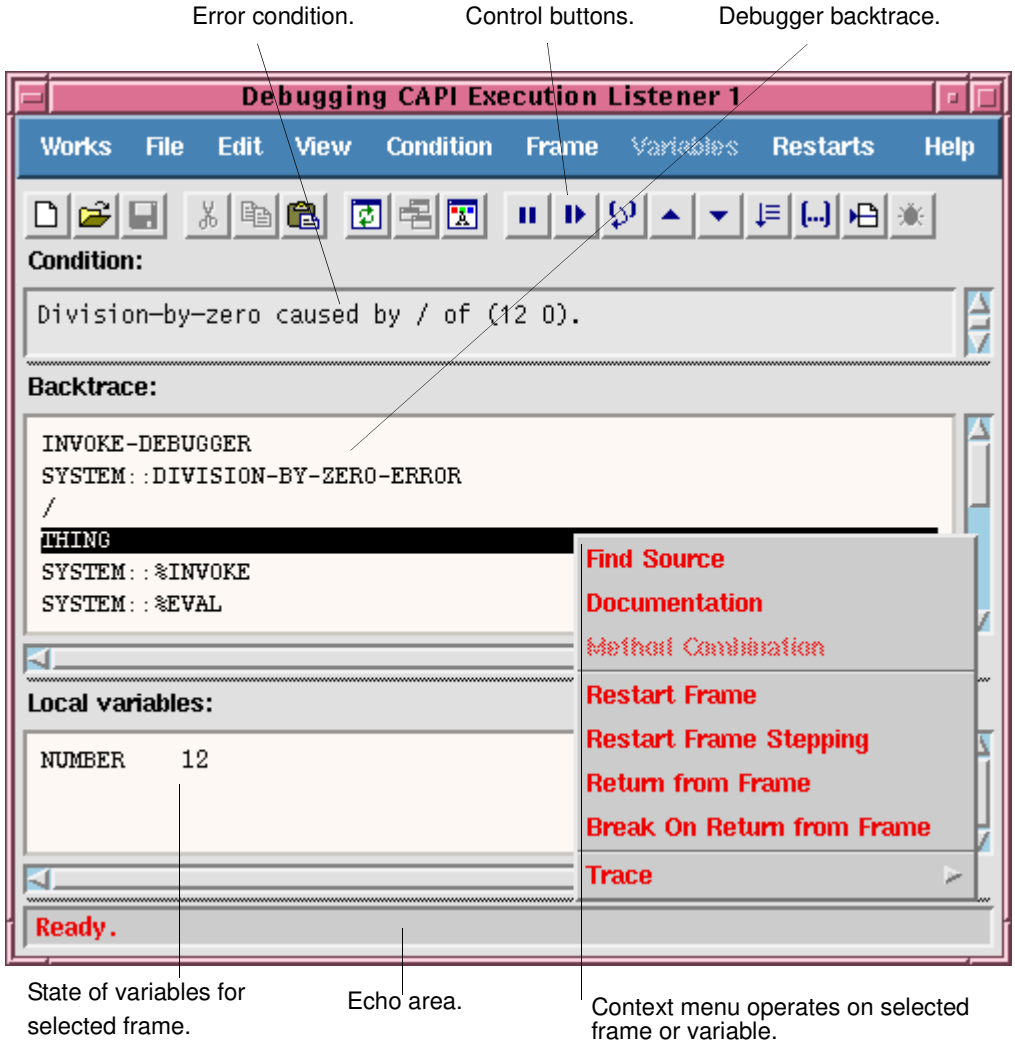
The colored circular nodes in the backtrace tree have these meanings:

Yellow	A normal lexical variable
Red	A closure variable (either from an outer scope or used by an inner scope)
Purple	A special variable

10.1.2.2 Frames and Variables in two lists

When the **Frames and Arguments** preference has the value **Two list-panels**, the Debugger appears as shown in below.

Figure 10.3 Debugger tool with two list-panels



Each item in the **Backtrace:** list represents a stack frame associated with a function call. You can operate on the frame as described in “Backtrace area” on page 119.

A second list titled **Local variables:** shows the local variables of the frame which si selectd in the **Backtrace:** list. You can operate on the variables similarly to the backtrace tree: double click on a variable to inspect it or use the commands in the **Variables** menu, which is also available as the context menu.

Note: with **Two list-panels**, only the local variables of the current frame are displayed.

10.1.3 Toolbar buttons

At the top of the debugger tool is a row of buttons, as described below. Press:

- to break the current execution.
- to return from the debugger and invoke the continue restart.
- to return from the debugger and invoke the abort restart.
- to select the previous stack frame in the backtrace area.
- to select the next stack frame in the backtrace area.
- to print the backtrace in the Listener.
- to print the variable bindings of the current frame in the Listener.
- to find the source code for the current stack frame.

If you hold the mouse cursor stationary over any button for about one second, then help text appears that identifies the button.

10.2 What the Debugger tool does

The Debugger tool provides a number of important facilities for inspecting programs.

Common Lisp, like most programming languages, uses a stack to store data about programs during execution. The Debugger tool allows you to inspect and change this stack to help get your programs working properly.

You can use it to trace backwards through the history of function calls on the stack, to see if the program behaves as expected, and locate points at which things have gone wrong.

You can also inspect variables within those functions, again to verify that the program is doing what is expected of it.

The Debugger tool also allows you to change variables on the stack. This is useful when testing possible solutions to the problems caused by a bug. You can run a bugged program, and then test fixes within the Debugger tool by altering values of variables, and then resume execution of the program.

10.3 Simple use of the Debugger tool

When you enter the Debugger tool, the Condition area displays a message describing the error. The **Restarts** menu lists a number of *restart* options, which offer you different ways to continue execution.

1. For example, type the name of a variable which you know is unbound (say **`fubar`**) at the Listener prompt.
2. Press in the Listener toolbar or choose **Debug > Start GUI Debugger** to enter the Debugger tool.
3. Select the **Restarts** menu to display the options available.

   A number of restarts are displayed that offer you different ways in which to proceed. These are the same options as those displayed at the command line debugger before you invoked the debugger tool.

   Two special restarts can be chosen: the *abort* and *continue* restarts. These are indicated by the prefixes **(abort)** and **(continue)** respectively. As a shortcut, you can use the **Abort** or **Continue** toolbar buttons to invoke them, instead of choosing the appropriate menu command.

   In the case of the continue restart, different operations are performed in different circumstances. In this example, you can evaluate the form again. If you first set the variable to some value, and then invoke the continue restart, the debugger is exited.
4. In the Listener, set the value of **`fubar`** as follows:

   ```
   (setq fubar 12)
   ```
5. Press **Continue** in the debugger tool.

The debugger tool disappears, and the command line debugger is exited in the Listener, and the value 12 is returned; the correct result if the variable had been bound in the first place.

You can also press **Abort** to invoke the abort restart. This restart always exits the current level of the debugger and returns to the previous one, ignoring the error which caused the present invocation of the debugger.

In general, you should use the continue restart if you have fixed the problem and want to continue execution, and the abort restart if you want to ignore the problem completely and stop execution.

10.4 The stack in the Debugger

As already mentioned, the debugger tool allows you to examine the state of the *execution stack*, which is shown in the Backtrace area. This area consists of a sequence of *stack frames*. A stack frame is a description of some part of a program, or something relating to the program, which is packaged into a block of memory and placed on the stack during program execution. These frames are not directly readable without the aid of the debugger.

There can be frames on the stack representing active function invocations, special variable bindings, restarts, and system-related code. In particular, the execution stack has a *call frame* for each active function call. That is, it stores information describing calls of functions which have been entered but not yet exited. This includes information such as the arguments with which the functions were called. By default, only call frames for active function calls are displayed in the Backtrace area. See Section 10.9 on page 129 for details of how to display other types of call frame.

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 name of a function, and the names and values of its arguments, and local variables.

10.5 An example debugging session

To better understand how you can make use of the debugger, try working through the following example session. In this example, you define the factorial function, save the definition to a file on disk, compile that file and then call the function erroneously.

1. Choose **File > New** or click on .

   A new file is created and displayed in the Editor. If you have not already invoked the Editor, it is started for you automatically.

2. In the new file, define the function **fac** to calculate factorial numbers.

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

3. Choose **File > Save** or click on and enter a filename when prompted.

4. Choose **File > Compile and Load** to compile the file and load the resulting fasl file.

   The Editor switches to the output view while compilation takes place. When prompted, press **Space** to return to the text view. The **fac** function is now defined and available for you to use.

5. In the Listener, call **fac** erroneously with a string argument.

```
(fac "turtle")
```

   LispWorks notices the error: The arguments of **=** should be numbers, and one of them is not.

6. Choose **Debug > Start GUI Debugger** or press to invoke the Debugger tool.

   Take a moment to examine the backtrace that is printed in the Backtrace area.

7. Starting from the selected frame, expand or select the next three frames in the Backtrace area in turn to examine the state of the variables which were passed to the functions in each call frame. Pay particular attention to the **fac** function.

   The error displayed in the Condition box informs you that the **=** function is called with two arguments: the integer 1 and the string “turtle”. Clearly, one of the arguments was not the correct type for **=**, and this has caused entry into the debugger. However, the arguments were passed to **=** by **fac**, and so the real problem lies in the **fac** function.

   In this case, the solution is to ensure that **fac** generates an appropriate error if it is given an argument which is not an integer.

8. Double-click on the line **FAC** in the Backtrace area of the debugger tool.

The Editor appears. The subform within the definition of **fac** which actually caused the error is highlighted. Double-clicking on a line in the Backtrace area is a shortcut for choosing **Frame > Find Source** or using the button. If the Debugger can find the erroneous subform, this is highlighted, otherwise the definition itself is highlighted if it can be found.

9. Edit the definition of the **fac** function so that an extra **if** statement is placed around the main clause of the function. The definition of **fac** now reads as follows:

```
(defun fac (n)
  (if (integerp n)
      (if (= n 1) 1
        (* n (fac (- n 1))))
    (print "Error: argument must be an integer")))
```

The function now checks that the argument it has been passed is an integer, before proceeding to evaluate the factorial. If an integer has not been passed, an appropriate error message is generated.

10. Choose **File > Save** and **File > Compile and Load** again, to save, recompile and load the new definition.

11. Click on the **Abort** button in the debugger tool, to destroy the tool and return the Listener to the top level loop.

12. In the Listener, type another call to **fac**, once again specifying a string as an argument. Note that the correct error message is generated. You will see it twice, becase **fac** prints the message and then the Listener prints the return value of **fac**.

This next part of the example shows you how you can use the various restarts which are listed as commands in the **Restarts** menu.

1. Call **fac** again with a new argument, but this time type the word **length** incorrectly.

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

2. Choose **Debug > Start GUI Debugger** or press to invoke the debugger tool.

You can spot immediately what has gone wrong here, so the simplest strategy is to return a value to use.

3. Choose **Restarts > Return some values from the form (LEGNTH "turtle")**.

You are prompted for a form to be evaluated.

4. Enter `6` in the dialog and press OK. This is the value that would have been returned from the correct call to `(length "turtle")`.

Having returned the correct value from `(length "turtle")`, `fac` is called with the correct argument and returns the value `720`.

10.6 Performing operations on the error condition

You can perform operations on the error condition that caused entry into the debugger using the commands available in the **Condition** menu.

The standard action commands are available in the **Condition** menu. For more details about these commands, see Section 3.7 on page 35.

Choose **Condition > Report Bug** to generate a bug report template.

10.7 Performing operations on stack frames

Any frame in the Backtrace list can be operated on using commands in the **Frame** menu. This menu is also available as a popup from the backtrace area itself. The commands available allow you to operate on the function displayed in the selected frame.

10.7.1 Source location, documentation and method combination

Choose **Frame > Find Source** to search for the source code definition of the object pointed to by the current frame. If it is found, the file is displayed in the Editor: the cursor is placed at the start of the definition or at the subform which cause the error, if known. The form is highlighted. See Chapter 12, “The Editor” for an introduction to the Editor.

Choose **Frame > Documentation** to display the Common Lisp documentation for the object pointed to by the current frame, if any exists. Note that this is the result of the Common Lisp function `documentation`, not the supplied manuals. It is printed in a special Output Browser window.

Choose **Frame > Method Combination** to display a Generic Function Browser tool in the Method Combinations view for the arguments in the selected frame. This command is only available when the selected frame is a call to a standard method. See "Examining information about combined methods" on page 201 for information about using the Method Combinations view.

10.7.2 Restarts and returning from the frame

Choose **Frame > Restart Frame** to continue execution from the selected restart frame. The action that is taken when choosing this command is printed with each restart frame in the Backtrace area. Note that restart frames must be listed for this command to be available: see "Configuring the call frames displayed" on page 129 for details.

Choose **Frame > Restart Frame Stepping** to step through execution from the selected restart frame. This frame becomes the active frame in a Stepper tool. See Chapter 25, "The Stepper" for information about using the Stepper tool.

Choose **Frame > Return from Frame** to resume execution from the selected frame. A dialog prompts for a value to return from the selected frame. Previously entered values are available via a dropdown in this dialog. This option allows you to continue execution smoothly after you have corrected the error which caused entry into the debugger.

Choose **Frame > Break On Return from Frame** to trap execution when it returns from the selected frame. This command prints a message telling you that the trap has been set, and when Lisp returns from the frame it calls **`break`**, allowing you to enter the debugger again.

10.7.3 Tracing the function in the frame

Choose **Frame > Trace** to display the standard Trace menu. This lets you trace the function in the selected frame in a variety of ways: see "Tracing symbols from tools" on page 40 for details.

10.8 Performing operations on frame variables

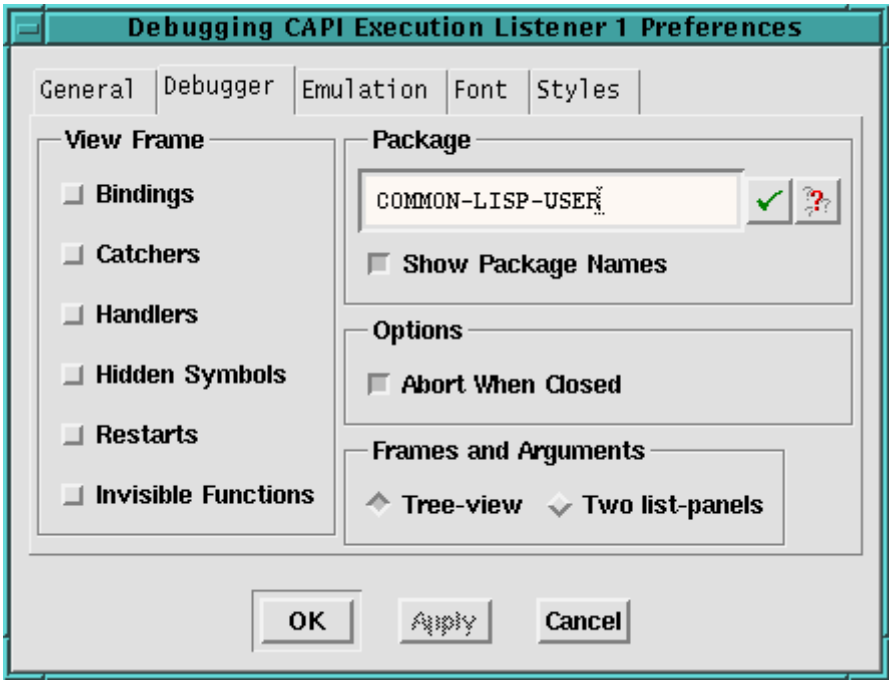
You can perform operations on a variable selected in the Variables area by the standard action commands which are available in the **Variables** menu or from the context menu on the variables list itself. For more details about these commands, see Section 3.7 on page 35.

Choose **Variables > Set...** to set the value of a variable selected in the Variables area. A dialog prompts you to enter a form which is evaluated to yield the new value for the variable. Previously entered forms are available via a dropdown in this dialog. The Common Lisp variable * is bound to the current value of the variable in the frame.

10.9 Configuring the debugger tool

You can control the behavior and appearance of the debugger using the Debugger Preferences dialog, which appears when you choose **Works > Tools > Preferences...** from the debugger.

Figure 10.4 Debugger Preferences dialog



10.9.1 Configuring the call frames displayed

By default, the call frame for each active function call in the backtrace is listed in the Backtrace area. There are a number of other types of call frame which are hidden by default. Display call frames of these types by selecting them in the View Frame panel of the debugger Preferences dialog.

Bindings Displays all the binding frames in the Backtrace list.

Catchers Lists the catch frames in the Backtrace list.

Handlers Lists the handler frames in the Backtrace list.

Hidden Symbols Lists any hidden symbols in the Backtrace list.

Restarts Lists all the restart frames in the Backtrace list. Each restart frame is listed, with the restart action to be taken given in brackets. To restart execution at any restart frame, select the frame, and choose **Debug > Frame > Restart Frame**.

Invisible Functions

Lists all invisible frames (such as the call to the error function itself) in the Backtrace list.

Note that all these commands can be toggled: choosing any command switches the display option on or off, depending on its current state. By default, all the options are off when the debugger is first invoked.

10.9.2 Displaying package information

As with other tools, you can configure the way package names are displayed in the debugger tool in the Package box of the Debugger Preferences dialog.

Check **Show Package Names** to turn the display of package names in the Backtrace and Variables lists on and off.

Specify a package name in the text box to change the process package of the debugger tool. You can use *completion* to reduce typing: click on to which allows you to select from a list of all package names which begin with the partial input you have entered. See “Completion” on page 46 for detailed instructions.

By default, the current package is the same as the package from which the error was generated.

10.9.3 Behavior on closing the Debugger

By default, when you close the Debugger window it attempts to abort, that is to call the abort restart.

Uncheck the **Abort When Closed** option only if you want to turn off this behavior.

10.9.4 Frames and variables display

To choose to view frames and variables in two lists rather than one tree, select the value **Two list-panels** in the **Frames and Variables** option.

11

The Tracer

11.1 Introduction

The Tracer tool is a debugging aid which gives you an interface to the LispWorks **`trace`** facilities. These allow you to follow the execution of particular functions and help you identify where errors occur during execution.

To create a Tracer, choose **Works > Tools > Tracer** or click in the Podium. Alternatively, a tracer can be created or displayed from within many other tools by choosing the command **Trace > Show in Tracer** in any menu whose commands operate on a traceable symbol.

The Tracer has three views:

- The Trace State view allows you to trace and untrace functions and change trace options for each function.
- The Output Data view records all tracing events in a tree structure and allows you to examine the arguments and results of each function call.
- The Output Text view shows all tracing events in textual format.

11.2 Tracing and Untracing functions

The Trace State view has a Trace pane where you can enter a function name. Press **`Return`** or click the button to trace that function.

The Traced Functions pane shows the list of functions that are currently traced. When some functions are selected, the **Function** menu contains the standard commands described in “Performing operations on selected objects” on page 35. As with other tools, choose **Edit > Select All** and **Edit > Deselect All** to select and deselect all the functions listed in the Traced Functions area.

The Selected Options area shows the trace options for a function selected in the Traced Functions pane. The trace options allow you to restrict or expand upon the information printed during a trace and can be modified by double-clicking on the item in the Traced Functions pane which raises the Trace Options dialog. For information about the trace options, see the section "Tracing options" in the *LispWorks User Guide*. Note that the options only apply to the first selected function. Each traced function has its own, independent, set of options.

The **Tracing Enabled** button can be used to turn all tracing off, whilst retaining the tracing state, and switch it back on again.

The **Untrace** button untraces the functions selected in the Traced Functions pane.

The **Untrace All** button untraces all functions.

In addition, the Tracer tool will track changes to the set of traced functions that are made from other tools, for example calls to the macros **`trace`** and **`untrace`** or the **Trace** submenu described in “Tracing symbols from tools” on page 40.

11.2.1 Tracing methods

You can trace methods (primary and auxiliary) within a generic function by entering the method dspec. For example, enter

```
(method my-function :before (integer))
```

in the Trace pane to trace the **`:before`** method of the generic function **`my-function`** that specializes on the class **`integer`**.

11.3 Examining the output of tracing

When you call a function that is traced, LispWorks collects information about the arguments it was called with and the values that it returned. This information is printed to the trace output stream, which might be the Listener or the Background Out-

put. In addition, if a Tracer tool is on the screen, the information is shown in its Output Text view and collected in its Output Data view in a tree format.

11.3.1 The Output Data view

Each call is a node in the tree with a λ icon and has three kinds of subnode:

- The subnode marked with a ⋙ shows the arguments passed to the function. Expanding this node shows each argument with its name (if known). Double-clicking on the arguments node or one of the arguments shows that argument in the Inspector.
- The subnode marked with a ⋘ shows the values returned from the function. Expanding this node shows each argument with its name (if known). Double-clicking on the arguments node or one of the arguments shows that argument in the Inspector.
- Any subnodes marked with a λ show calls to traced functions within the parent function.

You can collapse the tree by clicking on the toolbar button.

11.3.2 The Output Text view

This simply displays the textual trace output.

11.3.3 Clearing and restoring the Output

You can clear the trace output from both the Output views by clicking on the ✕ toolbar button.

You can restore the last cleared output by clicking on the ↶ toolbar button.

11.4 Example

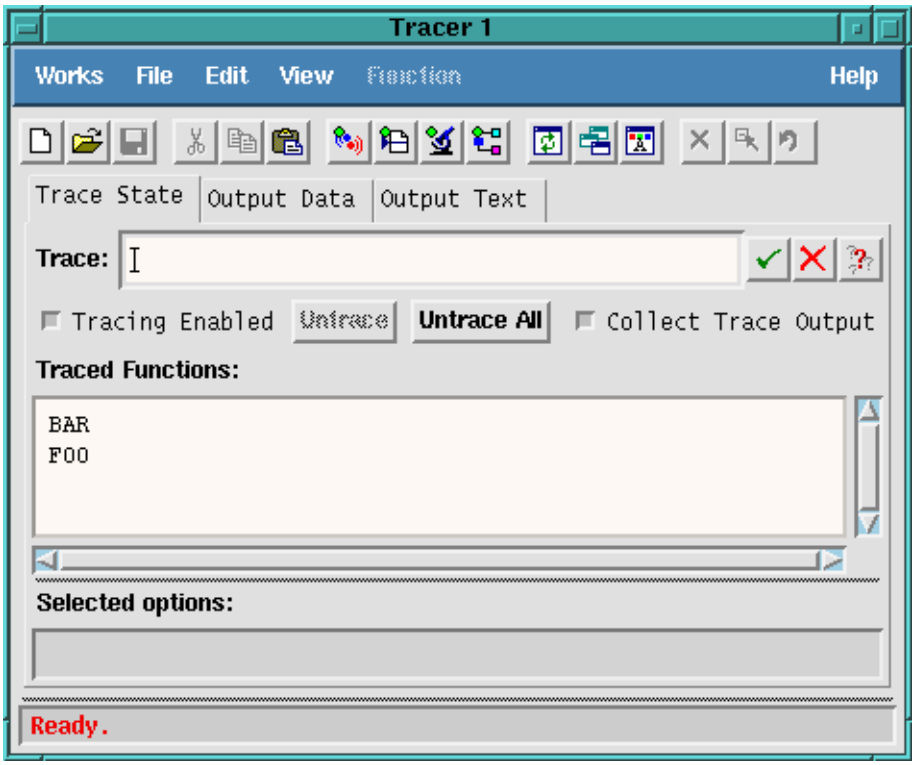
This section shows an example of tracing two functions and examining the output.

Define the following functions

```
(defun foo (x y) (bar y x))

(defun bar (x y) (values (vector x y) (list y x)))
```

in a Listener and start the Tracer tool. The trace these functions by entering **`foo`** into the Trace pane of the Tracer and pressing **`Return`** or clicking the ✓ button. Do the same for **`bar`**.

Figure 11.1 The Trace State view showing **`bar`** and **`foo`**



Then call

```
(foo 100 200)
```

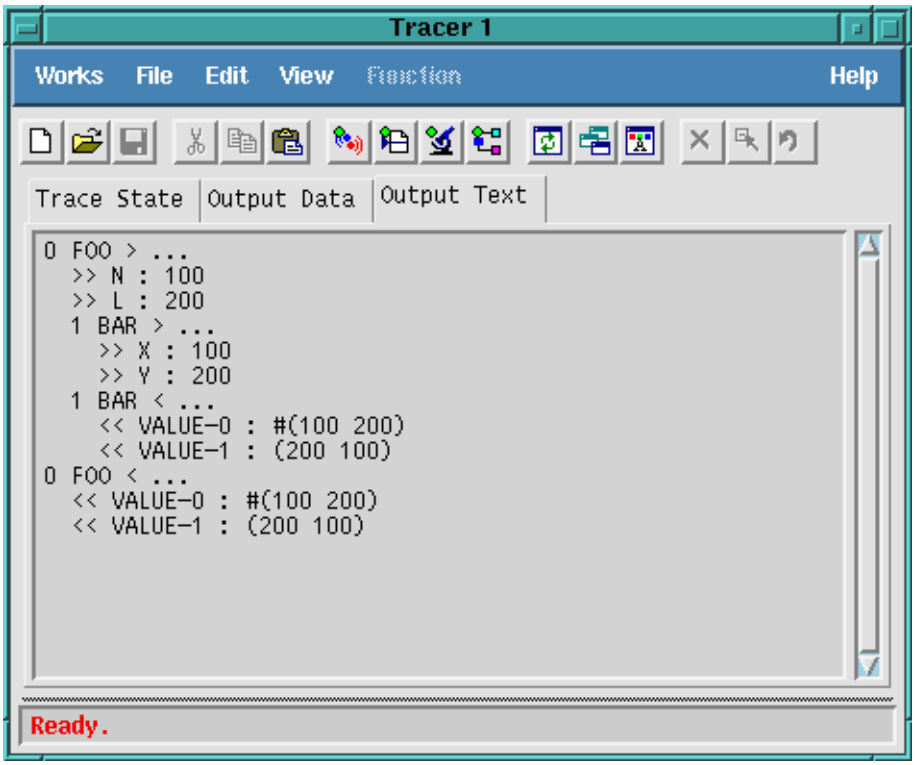
in the Listener. You will see output something like this printed in the Listener.

```
CL-USER 1 > foo 100 200
0 FOO > ...
  >> X : 100
  >> Y : 200
  1 BAR > ...
    >> X : 200
    >> Y : 100
  1 BAR < ...
    << VALUE-0 : #(200 100)
    << VALUE-1 : (100 200)
0 FOO < ...
  << VALUE-0 : #(200 100)
  << VALUE-1 : (100 200)
#(200 100)
(100 200)

CL-USER 2 >
```

Note: the format of the output is affected by the value of **`*trace-verbose*`**.

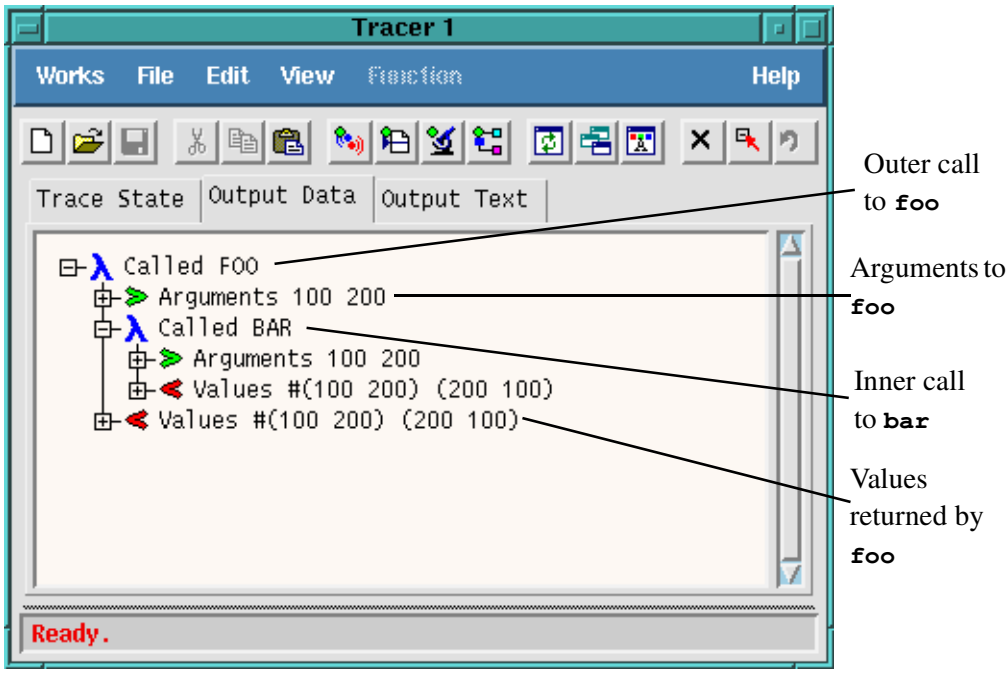
Now switch to the Output Text view of the Tracer and you will similar output.

Figure 11.2 The Output Text view



Now switch to the Output Data view of the Tracer, which will looks like this

Figure 11.3 The Output Data view



The node that is labeled **Arguments 100 200** contains the arguments to the function **`foo`**. Double-click on this node to show those arguments in an Inspector.

The first node that is labeled **Values #(200 100) (100 200)** contains the values returned by **`bar`**. Expand this node to reveal the two values. Double-click on one of the values nodes to inspect it. You can also see that these values were in turn returned by **`foo`**, as shown by the second node that is labeled **Values #(200 100) (100 200)**.

12

The Editor

The environment has a text editor which is designed specifically to make writing Lisp source code easier. By default it emulates the GNU EMACS text editor, and you should refer to the *LispWorks Editor User Guide* supplied with your software, for a full description of the extensive range of functions and commands available. It can also emulate a KDE/Gnomestyle text editor.

The Editor features a comprehensive set of menus, as well as a number of different views, and its interface is consistent with the other tools in the environment. This chapter gives a complete description of these aspects of the Editor, as well as giving you a general overview of how the Editor is used. If you have not used EMACS before, this chapter tells you all you need to know to get started.

The Editor is integrated with the other tools and offers a wide range of operations. The most commonly used of these can accessed using menu commands. The full range of editor commands is accessed via the keyboard commands described in more detail in the *LispWorks Editor User Guide*. These operations range from simple tasks such as navigating around a file, to more complex actions which have been specifically designed to ease the task of writing Lisp code.

By becoming familiar with the menu commands available, you can learn to use the Editor effectively in a very short space of time, before moving on to more advanced operations.

Like many other tools, the Editor offers a number of different views, which you can switch between using the tabs at the top of the Editor window. Unlike other tools, one view in particular is used more often than any other.

- The text view is the most commonly used view in the Editor. This lets you read and edit text files which are stored in your filesystem.
- The output view shows output messages. Compiler messages are highlighted and you can easily locate the source code that generated them.
- You can edit many different files at once in the same Editor. The buffers view provides a quick way of navigating between different files that you have open.
- The definitions view is a convenient way of seeing the classes, functions, macros, variables and so on that are defined in the current file.
- Files may contain many definitions. The find definitions view lets you search for particular definitions of interest across many files.

You can create an Editor using any of the following methods:

- Choose **Tools > Editor**. Notice that you are not actually editing a file immediately when you create an Editor like this.
- Choose **File > Open...**, or click on in the toolbar, and choose a filename in the dialog that appears.
- Choose **File > Recent Files** and choose a filename from the submenu that appears.
- Make the Listener the active window, and press **`Ctrl+X Ctrl+F`**. Type in the name of a file that you want to edit. If the file is not in the current directory, enter the full pathname.
- Choose the command **Find Source** (available on various menus, for example **Frame** in the Debugger tool), or click on to display source code in an Editor tool.

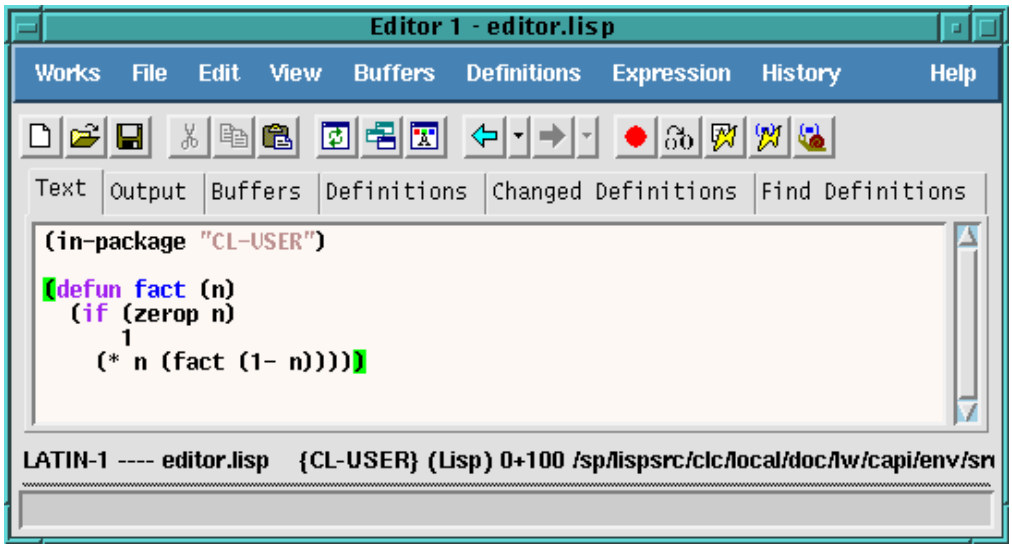
Note: this chapter assumes you are using the default EMACS emulation. Thus one way to open a file is with the keystrokes **`Ctrl+X Ctrl+F`** as described above. If you use KDE/Gnome keys, you would use instead the keystroke **`Ctrl+O`**.

You can always discover which key to use for a particular editor command, or conversely which command is invoked by a particular key. See “Help with editing” on page 185 for details.

12.1 Displaying and editing files

The text view is the default view in the Editor, and is the one which you will become most familiar with. In this view, a buffer containing the text of the current file is displayed, and you can move around it and change its contents as you wish, then save it back to the original file (assuming that you have permission to write to it). The text view is automatically displayed when you first invoked the Editor, and you can click on the Text tab to switch back to it from any other view. Figure 12.1 below shows an Editor in the text view with a file open.

Figure 12.1 Text view in the Editor



The text view has three areas, described below.

12.1.1 The toolbar

The Editor toolbar offers easy access to commands which operate on source code. In the text view it allows you to set breakpoints, and macroexpand, compile or evaluate code.

The Editor toolbar also contains the standard history toolbar. This is enabled in every view of the Editor tool.

12.1.2 The editor window

The editor window is the main part of the Editor. The text of the current file is shown in this area. A block cursor denotes the current position in the files in EMACS emulation. In KDE/Gnome editor emulation, a vertical line cursor appears in the active editor window. Text is entered into the file at this position when you type or paste.

To move the cursor to a particular point in the file, you can use any combination of the following methods:

- Position the cursor by moving the mouse pointer and selecting the point at which you want to place the cursor.
- If the file is too large to display all of it in the editor window, use the scroll bars to move up and down the file.
- Use any of the numerous keyboard commands that are available for navigating within a file.

If you are unfamiliar with the Editor, you can use the first two methods to begin with. As you become more familiar, you will find it is often quicker to use the keyboard commands described in the *LispWorks Editor User Guide*. Some of the most basic commands are also described in this chapter, in Section 12.8 on page 164.

12.1.3 The echo area

Underneath the editor window is an echo area, identical to the echo area in the other tools. This is used by the Editor to display status messages, and to request more information from you when necessary. The echo area is contained in every view in the Editor.

Whenever you invoke a command which requires further input (for instance, if you search a file for a piece of text, in which case you need to specify the text you want to

search for), you are prompted for that input in the echo area. Type any information that is needed by the Editor, and the characters you type are displayed ("echoed") in the echo area.

For many commands, you can save time by using *completion*. When you have partially specified input in the echo area, you can press a key (usually **Tab**, **?** or **Space**, depending on the command) and the Editor attempts to complete what you have typed. If it cannot complete your partial input uniquely, a window appears which lists all the possible alternatives and allows you to select the desired completion. See "Completion" on page 46 for detailed instructions.

For example, suppose you have three files in the current directory, **test1.lisp**, **test2.lisp** and **test3.lisp**, and you want to edit **test2.lisp** using keyboard commands. Type **Ctrl+X Ctrl+F**, then type **test** and press **Tab**. A list appears which shows all three files. To edit **test2.lisp**, double-click on the item marked **test2.lisp** in this list. For longer lists, the completion GUI helps you to quickly reduce the choice. See "Completion" on page 46 for details.

To see when completion is appropriate and when it is not, experiment by pressing the **Tab** key when typing in the echo area. As a rule, if there are a finite number of things you could meaningfully enter, then completion is appropriate. Thus, when opening a file already on disk, completion is appropriate (there is a finite number of files in the current directory). When specifying a string to search for, however, completion is not appropriate (you could enter any string).

12.1.4 Using keyboard commands

A full description of the keyboard commands available in the Editor is beyond the scope of this manual, and you are advised to study the *LispWorks Editor User Guide* to gain a full appreciation of the capabilities of the Editor. However, of necessity, certain basic keyboard commands are discussed in this chapter. See Section 12.8 on page 164 of this manual for a brief introduction to some of the most important ones. The menu commands available are described throughout the rest of this chapter.

As with other keyboard commands used in the environment, the keyboard commands used in the Editor are invoked by using a combination of the modifier keys **Control**, **Shift**, **Escape**, **Alt** and **Command** (not all of these are available on each platform), in conjunction with ordinary keys. Some of the commands available perform the same, or a similar task as a menu command.

Each keyboard command in the editor is actually a shortcut for an *extended editor command*. You can invoke any extended command by typing its command name in full, preceded by the keyboard command **Alt+X**. Thus, to invoke the extended command **Visit Tags File**, type **Alt+X visit tags file** followed by **Return**. Case is not significant in these commands, and completion (described in "Completion" on page 46) may be used to avoid the need to type long command names out in full. This method is often useful if you are not certain what the keyboard shortcut is, and there are many extended commands which do not have keyboard shortcuts at all.

Many of the keyboard commands described in this chapter and in the *LispWorks Editor User Guide* also work in the Listener. Feel free to experiment in the Listener with any of the keyboard commands that are described.

12.2 Displaying output messages in the Editor

As with several other tools, the Editor provides an output view which can be used to examine any output messages which have been generated by the environment. Click on the Output tab to switch to this view. See Chapter 21, "The Output Browser", for more information about this view.

12.3 Displaying and swapping between buffers

The contents of the editor window is the *buffer*. Technically speaking, when you edit a file, for example by **File > Open...**, its contents are copied into a buffer which is then displayed in the window. You actually edit the contents of the buffer, and never the file. When you save the buffer, for example by **File > Save**, its contents are copied back to the actual file on disk. Working in this way ensures that there is always a copy of the file on disk—if you make a mistake, or if your computer crashes, the last saved version of the file is always on disk, ensuring that you do not lose it completely.

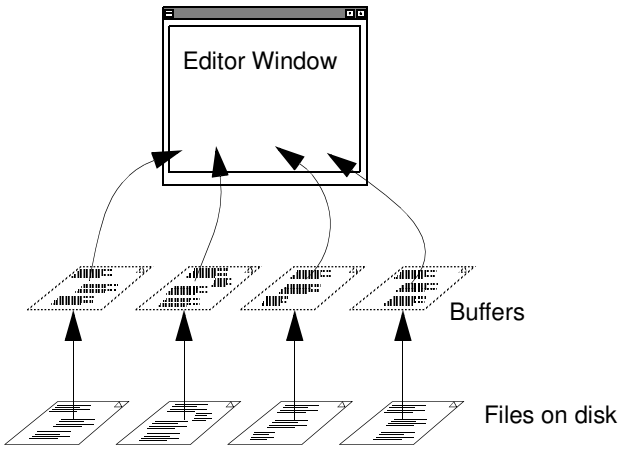
Because of this distinction, the term buffer is used throughout, when referring to the text in the window.

An Editor can only have one editor window, although there can be many buffers open at once. This means that you can edit more than one file at once, although only one buffer can be displayed at a time in the window—any others remain hidden.

When you close a buffer, for example with the menu command **File > Close** or the key **Ctrl+X K**, the buffer is removed. This is different to the command **Works > Exit > Window** which closes the window and does not affect the buffer.

The diagram below shows the distinctions between the window, buffers and files on disk.

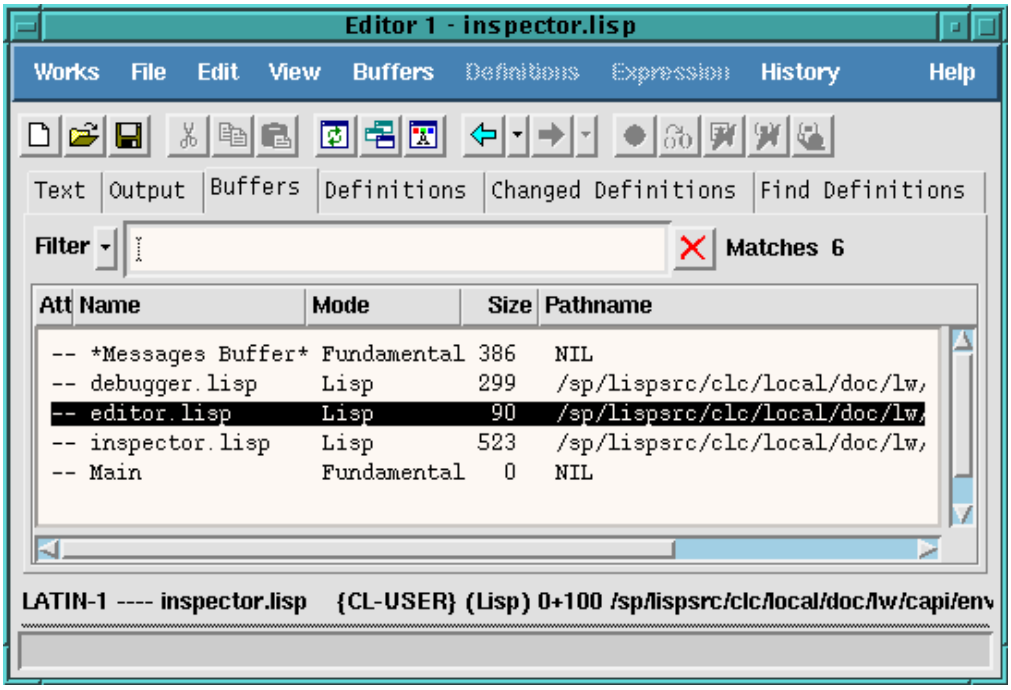
Figure 12.2 Distinctions between the window, buffers, and files on disk



The buffers view allows you to display a list of all the buffers that are currently open in the Editor, and gives you an easy way of navigating between them. Click on the

Buffers tab to switch to this view, or press **Ctrl+X Ctrl+B**. The Editor appears as shown in Figure 12.3 below.

Figure 12.3 Listing buffers in the Editor

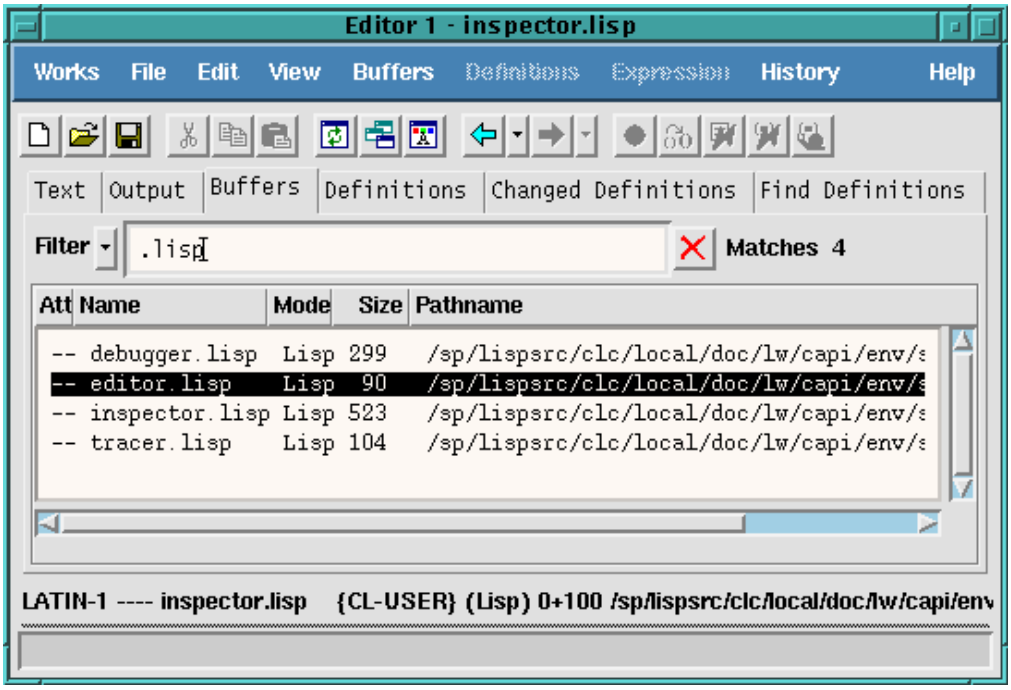


The buffers view has two areas, described below.

12.3.1 Filter area

You can use this area to restrict the number of buffers displayed in the Buffers area. For example you could display just the Lisp source files (that is, those with file type **`lisp`**) by entering **`.lisp`** as shown in Figure 12.4, page 149.

Figure 12.4 Filtering the buffers list in the Editor



See “Filtering information” on page 42 for details about how to use the Filter area in a tool.

12.3.2 Buffers area

Each item in the Buffers area list represents an editor buffer. Properties of the buffer such as its size (in bytes) and its mode are displayed. See the *LispWorks Editor User Guide* for information about editor modes.

Double-click on any buffer to display it in the Editor’s text view.

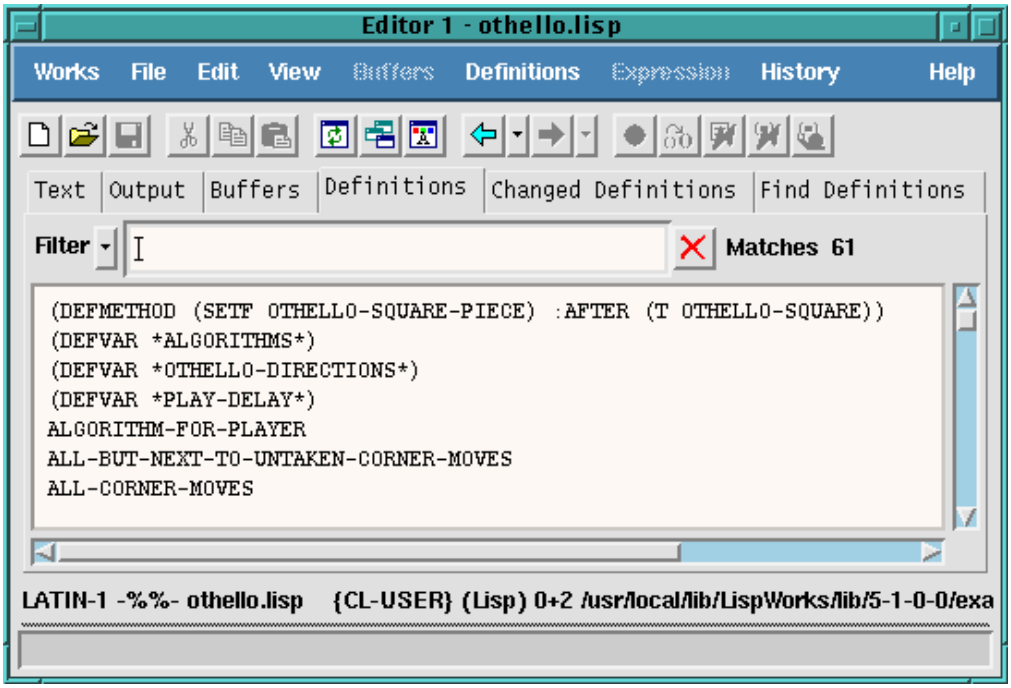
Buffers selected in the Buffers area can be operated on by commands in the **Buffers** menu, which is also available as the context menu. The associated files can be oper-

ated on by commands in the **File** menu. For example, to save multiple buffers, select them the Buffers area and choose **File > Save**. See “Using Lisp-specific commands” on page 178 for more details.

12.4 Displaying Common Lisp definitions

The definitions view lists all the Common Lisp definitions which can be found in the current buffer. Open a file containing several defining forms, such as the Othello game example in **`examples/capi/applications/othello.lisp`**. and then click on the Definitions tab. The Editor appears as shown in Figure 12.5 below.

Figure 12.5 Examining Common Lisp definitions in the Editor



The definitions view has two areas, described below.

12.4.1 Filter box

You can use this area to restrict the number of definitions displayed in the definitions area. See Section 3.11 on page 42 for details about how to use the Filter box in a tool.

12.4.2 Definitions area

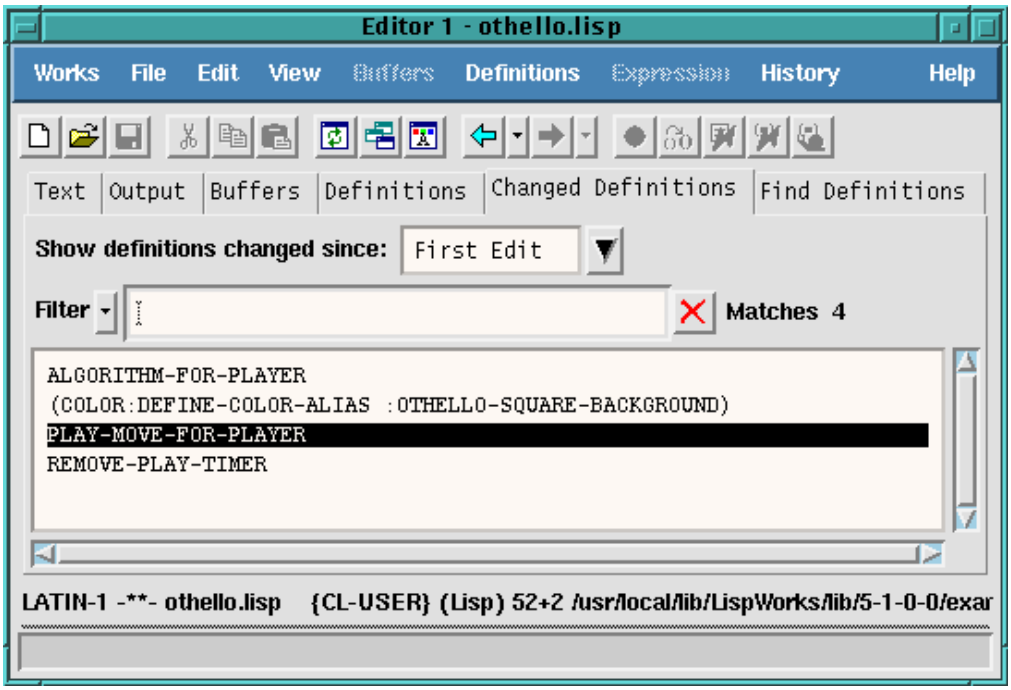
Double-click on any definition in this area to display its source code in the Editor's text view. Definitions selected in this area can be operated on using commands in the Editor's **Definitions** menu, which is also available as the context menu. See "Other facilities" on page 184 for complete details of the commands available.

12.5 Changed definitions

The Changed Definitions view allows you to see which definitions have been edited in the current session.

Edit some of the definitions in the Othello game example in **examples/capi/applications/othello.lisp** and then click on the Changed Definitions tab. The Editor appears as shown in Figure 12.6 below.

Figure 12.6 The Changed Definitions view in the Editor



Notice that the Changed Definitions view is similar to the Definitions view. The Editor's **Definitions** menu, and the filter box, can be used on definitions listed here in the same way as in the Definitions view.

12.5.1 Setting the reference point for changed definitions

The Changed Definitions view has an additional area "Show definitions changed since:". This allows you to change the reference point against which the current buffer is compared when computing the changes.

The reference point can be:

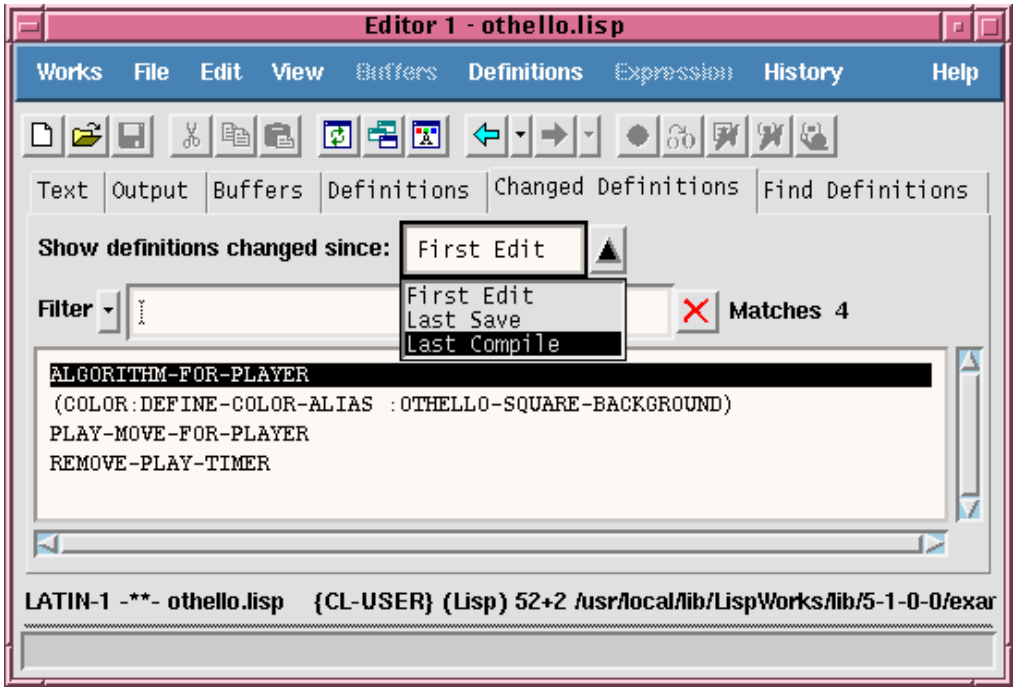
First Edit The state of the buffer just before you first edited it in the current LispWorks session. This is the initial reference point.

Last Save The state of the buffer when you last saved it to file

Last Compile The state of the buffer when you last compiled it.

Select from the **Show definitions changed since:** popup list to change the reference point.

Figure 12.7 Setting the reference point in the Changed Definitions view.



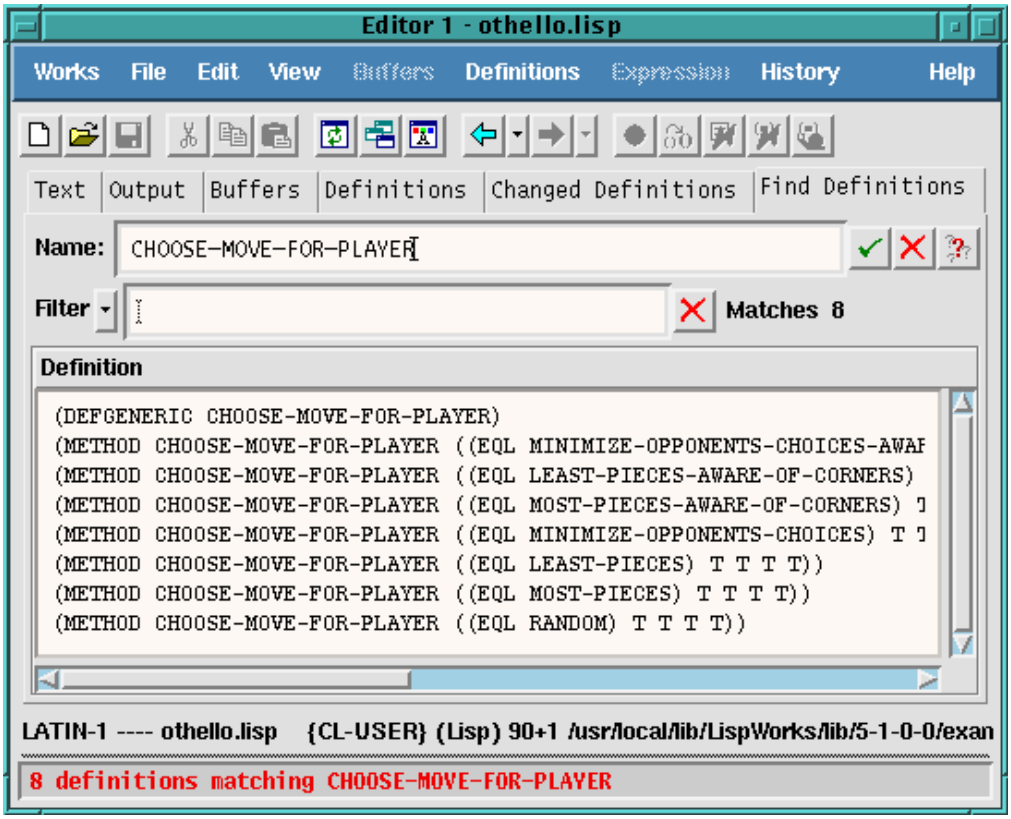
When you alter the reference point, the list of changed definitions is recomputed.

The list of changed definitions is computed using the editor command Buffer Changed Definitions. See the *Editor User Guide* for more information about this and related commands.

12.6 Finding definitions

Use the Find Definitions view to locate definitions recorded by the system with a given name. Firstly click on to ensure you have compiled the buffer displaying the Othello example. Then enter the name of the definition you are searching for in the name box and press **Return** or click on to display a list of matches together with their locations. Double-click on a match to display the source.

Figure 12.8 Displaying matches in the Find Definitions view



In addition, after using the editor command **Find Source** (bound to **Alt+.**) or other source location commands, you can invoke the Find Definitions view to display a complete list of the matches with the editor command **Alt+X View Source Search**.

Further, the option **Use Find Definitions list for more items than:** controls automatic use of this view, as described in “Automatic use of Find Definitions view” on page 158.

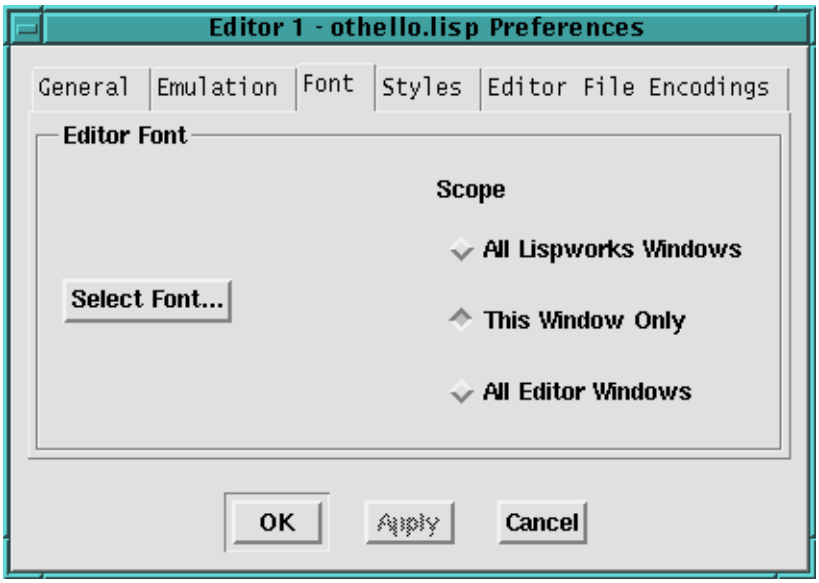
12.7 Setting Editor preferences

You can configure many aspects of the Editor tool, including:

- how items are listed in buffers and definitions views
- whether the Editor toolbar is displayed
- key input
- font
- the text styles used to highlight selected text, color code syntax, and so on
- the default encodings used for reading and writing files

To configure the Editor tool choose **Works > Tools > Preferences...** within an Editor to display the Editor Preferences dialog.

Figure 12.9 Editor Preferences dialog



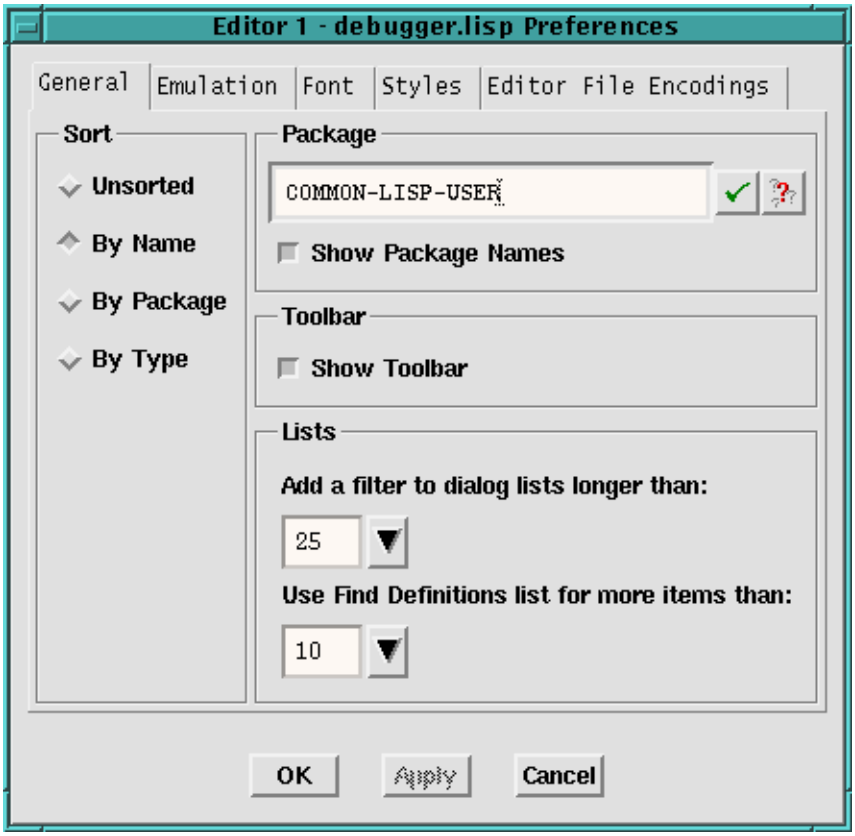
First specify your change in the Editor Preferences dialog. To make the change without dismissing the dialog, choose **Apply**. This is useful when experimenting with your configuration. To make the change and dismiss the dialog, choose **OK**.

Your change is preserved for your next LispWorks session.

12.7.1 Sorting items in lists

By default, items in the buffers and various definitions views are sorted alphabetically according to their name. The options in the Sort panel of the General tab in the Editor Preferences dialog let you change this.

Figure 12.10 The General tab of the Editor Preferences dialog



Unsorted	Leaves items in these lists unsorted. For views which list definitions, choosing this option lists definitions in the order in which they appear in the source code.
By Name	Sort according to the item name. This is the default setting.
By Package	Sort according to the buffer package or the package of the definition's name.
By Type	Sorts items according to the type of the definition, or the attributes of the buffer.

12.7.2 Displaying package information

As with many other tools, you can configure the way package names are displayed in the Editor. Because of the nature of this tool, you need to be a little more aware of the precise nature of these commands in order to avoid confusion. This information can be configured using the Package box of the Editor Preferences dialog shown in Figure 12.10.

Click **Show Package Names** to toggle display of package names in the main areas of the buffers and various definitions views.

Type a package name into the text field to change the current package in the Editor. You can use completion to reduce typing, by clicking which allows you to select from a list of all package names which begin with the partial input you have entered. See "Completion" on page 46 for detailed instructions. When you have entered the complete name, click the button to confirm the package name.

Note that this does *not* change the package currently displayed; it merely changes the Editor's notion of "where" it is in the environment, and this in turn affects the way symbols are printed in the buffers and various definitions views.

By default, the current package is **CL-USER**.

12.7.3 Controlling toolbar display

You can control whether Editor tools display the source operations and history toolbars by the option **Show Toolbar**, as described in "Toolbar configurations" on page 24.

12.7.4 Automatic filters on dialogs

The option **Add a filter to dialog lists longer than:** affects modal dialogs containing long lists. When the list is longer than the value of this option, the list has a filter, which you can use as described in "Filtering modal dialog completion" on page 49.

12.7.5 Automatic use of Find Definitions view

The option **Use Find Definitions list for more items than:** affects the behavior of source location commands such as the editor commands **`Find Source`** and **`Find Source for Dspec`**, and the menu command **Expression > Find Source**. When the number of source location results exceeds the value of this option, then the results are immediately displayed in the Find Definitions view of an Editor tool. This is particularly useful when you need to locate the definition of a particular CLOS method from the generic function name.

The Find Definitions view is described in "Finding definitions" on page 154.

12.7.6 Configuring the editor emulation

You can configure the editor to behave according to one of two pre-defined editor input styles (emulations) which determine how keyboard input is processed and other properties such as the shape of the input cursor. You can also set the cursor blink rate.

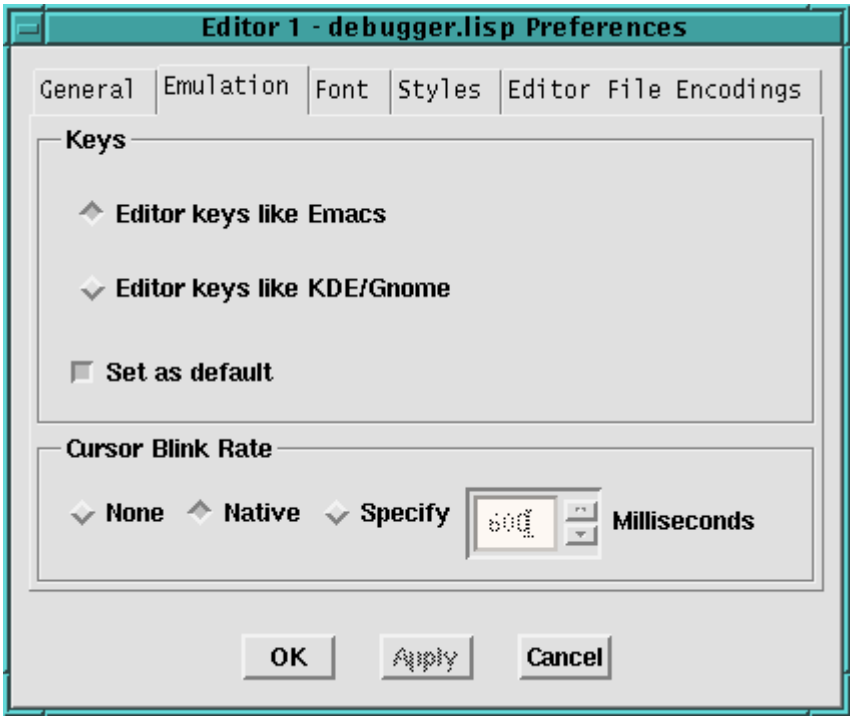
Note: The choice of emulation affects not only the Editor too, but also other LispWorks tools containing editors such as the Profiler.

12.7.6.1 Choosing the key input style

The Editor and other tools using **`capi:editor-pane`** offer two key input styles: EMACS emulation or KDE/Gnome emulation. By default, EMACS emulation is

used. To choose the alternate emulation, select the Emulation tab of the Editor Preferences dialog.

Figure 12.11 The Emulation tab of the Editor Preferences dialog



Note: In this and other manuals, the EMACS keys are generally given. For help with findings keys for editor commands, choose **Help > Editing > Command to Key**. Also see the files **`config/key-binds.lisp`** and **`config/msw-key-binds.lisp`** which contain the forms defining the keys for each input style.

12.7.6.2 Setting the cursor blink rate

By default the editor cursor blinks on and off at the usual rate for your computer.

To change the blink rate, select **Specify** in the Blink Rate area. Either scroll to choose the rate in Milliseconds, or enter an integer between 100 and 2000.

To stop the editor cursor from blinking, select **None** in the Blink Rate area.

12.7.6.3 Preserving your setting

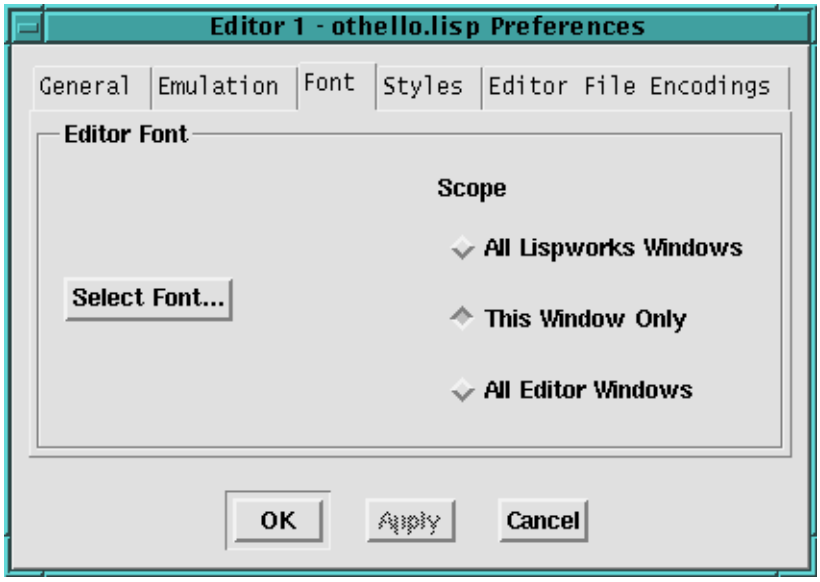
By default your setting is preserved for future use.

If you do not want the specified key input style to be registered for future Editors and other tools, uncheck **Set as default**.

12.7.7 Setting the editor font

To specify the font used in Editor and Listener windows and all other tools based on the editor, such as the Shell tool, click **Select Font...** in the Font tab of the Editor Preferences dialog.

Figure 12.12 The Font tab of the Editor Preferences dialog.



12.7.8 Setting the text style attributes

By default Common LispWorks uses a variety of text styles to:

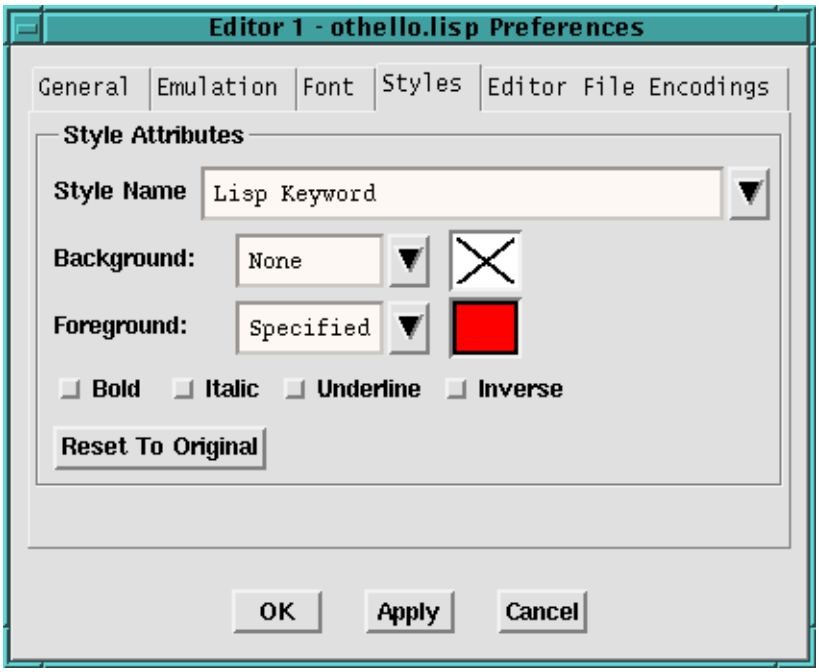
- highlight selected text
- distinguish interactive input in the Listener and Shell tools
- distinguish compiler messages in the Output tab or Output Browser

- make Lisp code more easily readable with syntax coloring
- indicate matching parentheses, easing the writing of correct Lisp forms

Note: The last two of these features operate only when the buffer is in Lisp mode.

To change the attributes of one or more text styles, select the Styles tab of the Editor Preferences dialog.

Figure 12.13 The Styles tab of the Editor Preferences dialog.



For example, to make Common Lisp symbols appear with red foreground rather than the default purple, first select Lisp Keyword in the **Style Name** list. Then select **Specified** alongside **Foreground** and double-click on the color area to the right. In the Color dialog that appears, choose the new color and click **OK**. Now click **OK** on the Editor Preferences dialog and see the change in the way your Lisp code is displayed. You may need to force the editor window to redisplay, for example by scrolling, to see the change take effect.

For each named style, the Foreground and Background each have exactly one of the following values:

None	No special formatting
Default	Platform-standard highlighting, as for selected text
Specified	The color specified is used.
Modified	The system generates a color which is usable for highlighting.

A large cross appears in the Foreground (Background) color area when **None**, **Modified** or **Default** is selected. This indicates that the color is not used for the Foreground (Background).

If you wish to turn off the highlighting of interactive input in the Listener and Shell tools, first select Interactive Input in the **Style Name** list. Then uncheck all the attributes and click **OK**.

To restore all styles to those in LispWorks as shipped, click **Reset to Original**.

Note: the foreground and background colors of windows are set via the system, not in LispWorks. To alter these colors on X11/Motif, specify the appropriate X resources for the application class Lispworks. The fallback resources are in the LispWorks library file **`app-defaults/Lispworks`**. You can specify X resources in a file in your home directory **`~/Lispworks`**.

12.7.8.1 Syntax styles

The text styles used in syntax coloring have these meanings and default appearance:

Table 12.1 Syntax styles

Style Name	Use	Default appearance
Region Highlight	The active region	Native highlight
Show Point	Matching parentheses	**`:green`** background
Interactive Input	Input in a Listener or Shell	Bold
Highlight	Editor help such as **`Describe Bindings`**	Bold

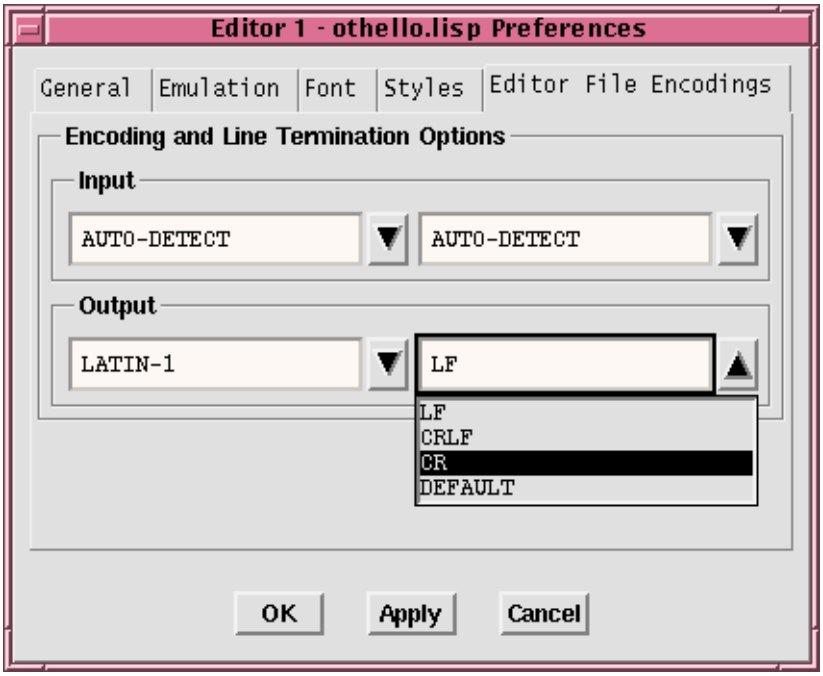
Table 12.1 Syntax styles

Style Name	Use	Default appearance
Completion	Dynamic and in-place completions. Transient.	Modified background
Lisp Function Name	Name in **`defun`**, **`defmacro`**, **`defmethod`** and **`defgeneric`** forms	**`:blue`** foreground
Lisp Comment	Comments and feature expressions	**`:firebrick`** foreground
Lisp Type	Name in **`deftype`** or other **`def...`** form, or lambda list keyword such as **`&optional`**	**`:forestgreen`** foreground
Lisp Variable Name	Name in **`defvar`** or **`defparameter`** forms	**`:darkgoldenrod`** foreground
Lisp String	A string literal	**`:rosybrown`** foreground
Lisp Keyword	**`defun`**, **`defmacro`** or other definer named **`def...`**	**`:purple`** foreground
Lisp Builtin	A keyword symbol	**`:orchid`** foreground

12.7.9 Setting the default encodings

The Editor has defaults for the encodings used when reading and writing files. For most uses these defaults will suffice. If you need to change either, select the Editor File Encodings tab of the Editor Preferences dialog.

Figure 12.14 The Editor File Encodings tab of the Editor Preferences dialog.



For example, to make the Editor write Carriage Return line-terminated files by default, select **CR** in the Line Termination Options for Output.

12.8 Basic Editor commands

This section deals with some of the most basic commands available in the Editor. It describes how to perform simple file management, how to move around a buffer, and tells you about some other more general commands available.

12.8.1 Opening, saving and printing files

When you first start up the Editor, the first thing you must do is open a file.

Use file extensions **`.lisp`** or **`.lsp`** for Common Lisp files. The Editor recognizes these extensions and places the buffer in Lisp mode. Lisp mode provides special features for use in Lisp editing, as described in “Lisp mode” on page 178.

You can create a new Lisp buffer by choosing **File > New** or clicking on . The new file is automatically in Lisp mode, and the buffer is called “Unnamed”. When you try to save this buffer, the Editor prompts you for a filename.

As you have already seen, you can open an existing file by choosing **File > Open...** or clicking on . A dialog appears from which you can select a file to edit.

To save a file, choose **File > Save** or click on . If the file has not been saved before (that is, if you created the file by choosing **File > New** and this is the first time you have saved the file), you are prompted for a directory and a filename.

You can also save a file by using the keyboard command **`Ctrl+X Ctrl+S`**.

If you want to make a copy of the file (save the file under a different name) choose **File > Save As...** and specify a name in the dialog that appears.

Choose **File > Revert to Saved** to revert back to the last saved version of the file. This replaces the contents of the current buffer with the version of that file which was last saved on disk. This command is useful if you make a number of experimental changes which you want to abandon.

As well as saving whole files to disk, you can save any part of a file to disk under a different filename. To do this:

1. Select a region of text by clicking and holding down the select mouse button, and dragging the pointer across the region of text you want to save. The text is highlighted as you drag the pointer across it.
2. With the text still highlighted, choose **File > Save Region As...**.
3. In the echo area, specify the name of a file to save the selected text to.

Note that the selected text is *copied* into the new file, rather than moved; it is still available in the original buffer.

To find out more about selecting regions of text, see “Marking the region” on page 171. To find out more about operating on regions of text, see “Using Lisp-specific commands” on page 178.

To print the file in the current buffer to your default printer, choose **File > Print...**. The printer can be changed or configured by choosing the **File > Printer Setup...** menu option.

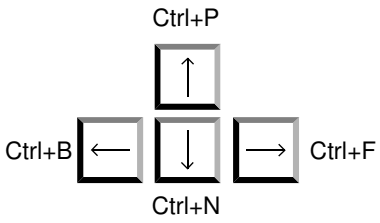
12.8.2 Moving around files

This section describes how you can move the cursor around the buffer. There are a variety of commands, allowing you to move sideways, up, or down by one character, or by a number of characters.

To move directly to any point in the buffer, position the pointer and click the left mouse button. If necessary, use the scroll bars to reveal sections of the buffer which are not visible in the window.

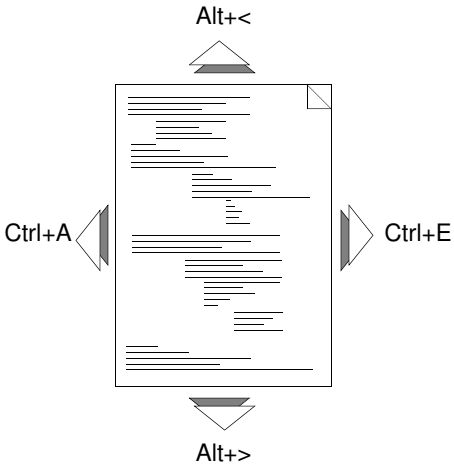
You can either use the arrow keys, or the keyboard commands shown below to move the cursor in any direction by one character.

Figure 12.15 Moving the cursor by one character



The keyboard commands below move to the beginning or end of the line, or the top or bottom of the buffer.

Figure 12.16 Keyboard commands for basic movement within an editor buffer



Press **Ctrl+V** or the **Page Down** key to scroll down one screenful of text.

Press **Esc V** or **Alt+V** or the **Page Up** key to scroll up one screenful of text.

You should ensure that you learn the keyboard commands described above, since they make navigation in a buffer much easier.

12.8.3 Inserting and deleting text

The editor provides a sophisticated range of commands for cutting text which are described in “Cutting, copying and pasting using the kill ring” on page 171. However, the two basic commands for deleting text which you should remember are as follows:

- To erase the previous character, use the **Backspace** key.
- To erase the next character, use **Ctrl+D** or the **Delete** key if available.

You can insert text into a buffer by typing characters, or by pasting (see “Cutting, copying and pasting using the kill ring” on page 171) or by inserting the contents of a file.

By default, when typing in a buffer, any characters to the right of the cursor are moved further to the right. If you wish to overwrite these characters, rather than preserve

them, press the **Insert** key. To return to the default behavior, just press the **Insert** key once more.

To insert the contents of one file into another, choose **File > Insert...**. A dialog appears so that you can choose a file to insert, and this is then inserted into the current buffer, starting from the current position of the cursor.

12.8.4 Using several buffers

As mentioned above, you can have as many buffers open at once as you like. Repeated use of **File > Open...** or **Ctrl+X Ctrl+F** just creates extra buffers.

Because the Editor can only display one buffer at a time, you can use either menu commands or keyboard commands to swap between buffers.

Each item in the **History > Items** submenu is an open buffer. To swap to a given buffer, choose it from the menu, and it is displayed in the editor window.

Alternatively, click on the Buffers tab to swap to the buffers view; see "Displaying and swapping between buffers" on page 146 for details.

To use the keyboard, type **Ctrl+X B**. You are prompted for the name of the buffer you wish to display. The last buffer you displayed is chosen by default, and is listed in the echo area in brackets, as shown below.

```
Select Buffer: (test.lisp):
```

To swap to the buffer shown in brackets, just press **Return**. To swap to another buffer, type in the name of that buffer. Remember that completion (press **Tab**) can help.

To close the buffer that is currently displayed, choose **File > Close**, or type **Ctrl+X K**.

- If you use **File > Close**, the current buffer is closed.
- If you use **Ctrl+X K**, you can close *any* buffer, not just the current one. Type a buffer name in the echo area, or press **Return** to close the current buffer.

Note: If you attempt to close any buffer which you have changed but not yet saved, a dialog appears, giving you the opportunity to cancel the operation.

To save all the buffers in the Editor, choose **File > Save All...**. A dialog appears which lists each modified buffer. By default, each buffer is selected, indicating that it is to be saved. If there are any buffers that you do not want to save, deselect them by clicking on them. The dialog has four buttons, as follows:

- Click **Yes** to save the selected buffers.
- Click **All** to save all the listed buffers.
- Click **No** to save none of the listed buffers.
- Click **Cancel** to cancel the operation.

This dialog is also displayed if there are any unsaved files when you exit the environment.

Sometimes you may find that being able to display only one buffer in the window simply does not give you enough flexibility. For instance, you may have several buffers open, and you may want to look at two different buffers at once. Or you may have a very large buffer, and want to look at the beginning and end of it at the same time.

You can do any of these by creating a new Editor window. Choose **Works > Clone** or press **`Ctrl+X 2`** or press the button. This creates a copy of your original Editor. The new Editor displays the same buffer as the original one.

- If you want to look at two different sections of this buffer at once, simply move to the section that you want to look at in one of the Editors.
- If you want to look at a different buffer, use the **History > Items** submenu or the keyboard commands described above to switch buffers.

Changes made to a buffer are automatically reflected across all editor windows—the buffer may be displayed in two different windows, but there is still only one buffer. This means that it is impossible to save two different versions of the same file on disk.

12.9 Other essential commands

Finally, there are three basic functions which you should add to your stock of familiar commands.

12.9.1 Aborting commands

To abort any command which requires you to type information at the echo area, type **`Ctrl+G`** at any point up to where you would normally press **`Return`**. For instance, if you type **`Ctrl+X Ctrl+F`** in order to open a file, and then decide against it, type **`Ctrl+G`** instead of specifying a filename.

If you are using KDE/Gnome editor emulation, press **`Esc`** to abort a command.

12.9.2 Undoing commands

If you choose **Edit > Undo** the last editor action performed is undone. Successive use of **Edit > Undo** revokes more actions (rather than undoing the last **Undo** command, as is the case with many other editors).

When using EMACS emulation you can undo via the EMACS keystroke **Ctrl+_**. Thus, to undo the last five words typed, press **Ctrl+_** five times.

If you are using KDE/Gnome editor emulation, press **Ctrl+Z** to undo.

12.9.3 Repeating commands

To perform the same command *n* times, type **Ctrl+U** *n* followed by the command you want to perform.

For instance, to move forward 10 characters, type **Ctrl+U 10 Ctrl+F**.

If you are using KDE/Gnome editor emulation, type **Ctrl+*** *n* followed by the command.

12.10 Cutting, copying and pasting using the clipboard

The Editor provides the standard methods of cutting, copying and pasting text using the clipboard. To select a region of text, click and hold down the select button, and drag the pointer across the region you want to select: the text is highlighted using the **Region Highlight** text style as you select it.

Choose **Edit > Select All** to select all the text in the buffer, and **Edit > Deselect All** if you want to deselect it.

Once you have selected a region use either of the following commands:

- Choose **Edit > Copy** to copy the region to the clipboard. This leaves the selected region unchanged in the editor buffer.
- Choose **Edit > Cut** to delete the region from the current buffer, and place it in the Common LispWorks clipboard. This removes the selected region from the buffer.

Choose **Edit > Paste** to copy text from the clipboard into the current buffer. The text is placed at the current cursor position.

These commands are also available from the context menu in the editor window, which is usually invoked by clicking the right mouse button.

The Editor also provides a much more sophisticated system for cutting, copying and pasting text, as described below.

12.11 Cutting, copying and pasting using the kill ring

The Editor provides a sophisticated range of commands for cutting or copying text onto a special kind of clipboard, known as the *kill ring*, and then pasting that text back into your Editor later on. There are three steps in the process, as follows:

- Marking a region of text.
- Cutting or copying the text in that region to place it in the kill ring.
- Pasting the text from the kill ring back into a buffer.

12.11.1 Marking the region

First of all, you need to mark a region of text in the current buffer which you want to transfer into the kill ring. There are two ways that you can do this:

- Select the text you want to copy or cut using the mouse. Click and hold down the Select mouse button, and drag the pointer across the region you want to mark.

  The selected text is highlighted using the **`Region Highlight`** text style.

- Using keyboard commands

  To mark the region with the keyboard, place the cursor at the beginning of the text you want to mark, press **`Ctrl+Space`**, and move the cursor to the end of the region you want to mark, *using keyboard commands to do so*. Unlike marking with the mouse, this does not highlight the region.

Because the Editor does not highlight the marked region when you use keyboard commands, a useful EMACS key to remember is **`Ctrl+X Ctrl+X`**. Pressing this exchanges the current cursor position with the start of the marked region and highlights the region. Press **`Ctrl+X Ctrl+X`** a second time to return the cursor to its original position and leave the region marked.

Press **Ctrl+G** (or **Esc** in KDE/Gnome emulation) to remove the highlighting in a region.

12.11.2 Cutting or copying text

Once you have marked the region, you need to transfer the text to the kill ring by either cutting or copying it.

Cutting text moves it from the current buffer into the kill ring, and deletes it from the current buffer, whereas copying just places a copy of the text in the kill ring.

- Choose **Edit > Cut** or press **Ctrl+W** to cut the text. In KDE/Gnome emulation the key is **Ctrl+X**.
- Choose **Edit > Copy** or press **Alt+W** to copy the text. In KDE/Gnome emulation the key is **Ctrl+C**.

Notice that these commands transfer the selected text to the Common LispWorks clipboard as well as the kill ring. This is so that the selected text can be transferred into other tools, or even into other applications.

UNIX Implementation Note: The selected text is also transferred to the UNIX clipboard.

12.11.3 Pasting text

Once you have an item in the kill ring, you can paste it back into a buffer as many times as you like.

- Press **Ctrl+Y** to paste the text in the kill ring back into the buffer. In KDE/Gnome emulation the key is **Ctrl+V**.

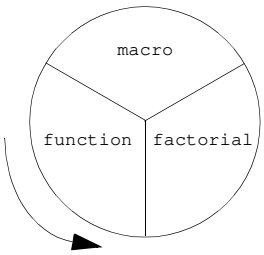
  Note that you must use the keyboard command if you wish to paste the item that is in the kill ring (as opposed to the item in the Common LispWorks clipboard).

With many editors you can only do this with one item at a time. The clipboard is only able to contain one item, and so it is the only one available for pasting back into the text.

However, the kill ring allows you to keep many items. Any of these items can be pasted back into your document at any time. Every time you cut or copy something, it is added to the kill ring, so you accumulate more items in the kill ring as your session progresses.

Consider the following example. In Figure 12.17, the kill ring contains three items; the words **factorial**, **function** and **macro** respectively.

Figure 12.17 Kill ring with three items

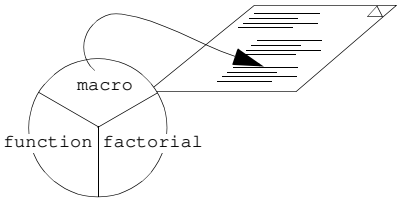


First, the word **factorial** was cut from the current buffer (this would remove it from the buffer). Next, the word **function** was copied (which would leave it in the buffer but add a copy of it to the kill ring), and lastly, the word **macro** was cut.

Note the concept of the kill ring rotating (this is why it is known as a ring). Every time a new item is added (at the top, in these figures), the others are all shunted around in a counter-clockwise direction.

Whenever you perform a paste, the current item in the kill ring—the word **macro** in this case—is copied back into the buffer wherever the cursor currently is. *Note that the current item is not removed from the kill ring.*

Figure 12.18 Pasting from the kill ring

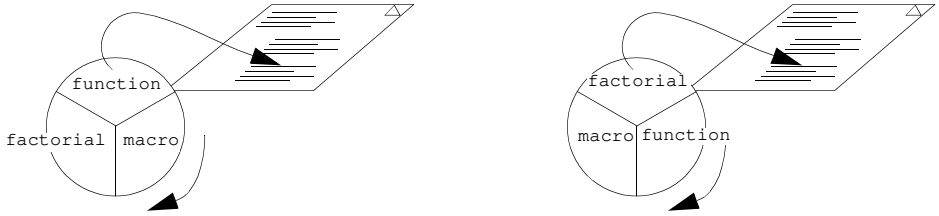


What you have seen so far does exactly the same thing as the standard clipboard. True, all three items have been kept in the kill ring, but they are of no use if you cannot actually get at them.

The EMACS key to do this is **Alt+Y** or **Esc Y**. This rotates the kill ring in the opposite direction—thus making the previous item the current one—and pastes it into the buffer in place of the item just pasted. In Figure 12.18, the word **macro** would be replaced with the word **function**.

You can use **Alt+Y** as many times as you like. For instance, if you actually wanted to paste the word **factorial** in the document, pressing **Alt+Y** would replace the word **function** with the word **factorial**.

Figure 12.19 Rotating the kill ring



If you pressed **Alt+Y** a third time, the kill ring would have rotated completely, and **macro** would have been the current item once again.

Note: You can never use **Alt+Y** without having used **Ctrl+Y** immediately beforehand.

Here is a summary of the way **Ctrl+Y** and **Alt+Y** work:

- **Ctrl+Y** pastes the current item in the kill ring into the buffer.
- **Alt+Y** rotates the kill ring back one place, and then pastes the current item into the buffer, replacing the previously pasted item.

12.12 Searching and replacing text

The Editor provides a wide range of facilities to search for and replace text. The examples below introduce you to the basic principles; please refer to the *LispWorks Editor User Guide* for a complete description of the facilities available.

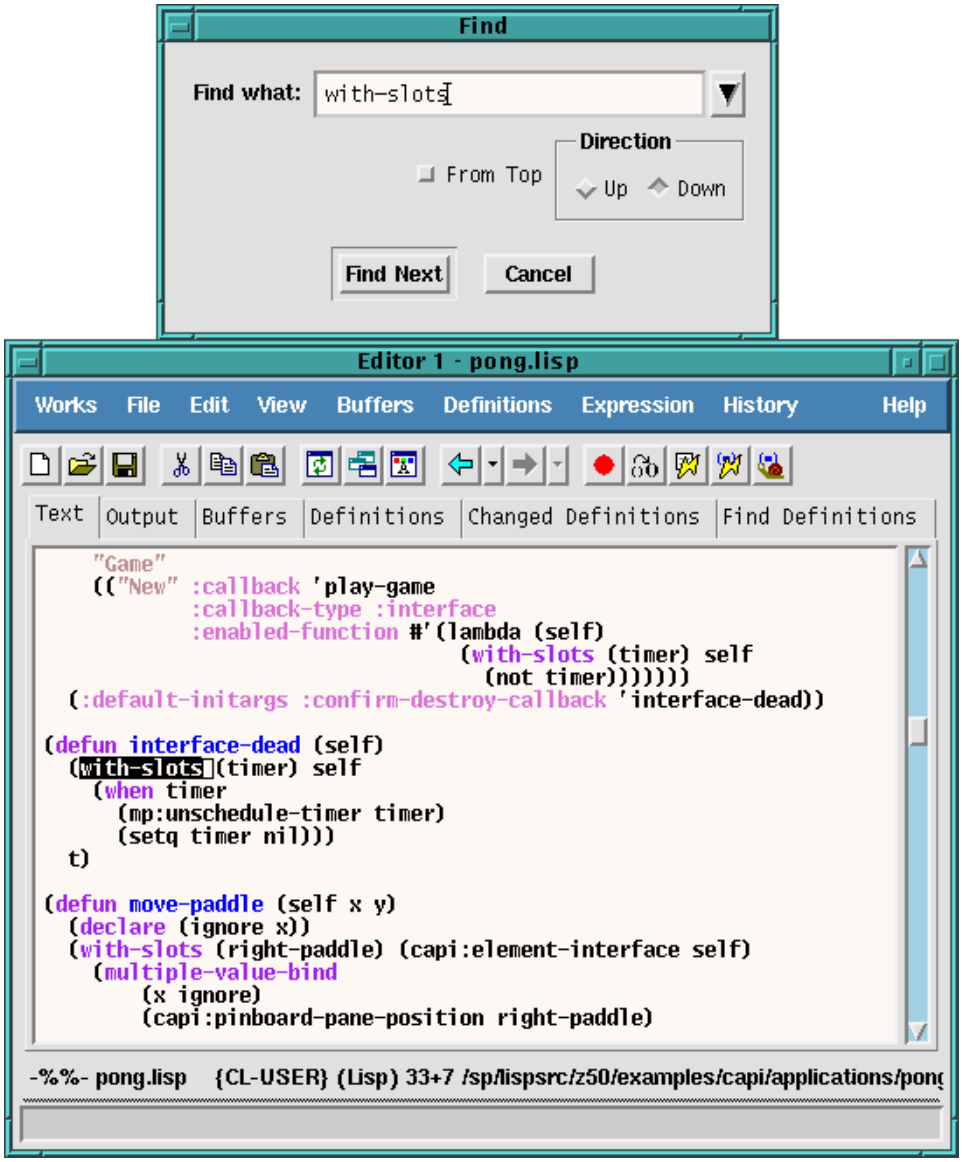
12.12.1 Searching for text

The simplest way of searching for text in a buffer is to use the commands available in the menu bar:

1. Choose **Edit > Find...** to search for text in the current buffer.
2. Type a string to search for in the dialog that appears.

3. Click the **Find Next** button.

Figure 12.20 Use of the Find dialog in the Editor



The cursor is placed immediately after the next occurrence in the current buffer of the string you specified. To search the buffer from the start, rather than the current point, check **From Top** and click **Find Next**. To search upwards, select **Up** in the Direction panel and click **Find Next**. To search again for a string that you previously searched for, select the string from the **Find what** list and click **Find Next**.

To dismiss the Find dialog, click **Cancel**.

After you have used the Find dialog, you can use **Edit > Find Next** to find the next occurrence of the last string for which you searched using the dialog, without raising the dialog again.

12.12.2 Incremental searches

Press **Ctrl+S** to perform an *incremental search* (in which every character you type further refines the search). A prompt appears in the echo area, asking you to type a string to search for. As soon as you start typing, the search commences.

Consider the following example: open the file **examples/capi/applications/othello.lisp**. You want to search for the word "defmethod" in the buffer.

1. Press **Ctrl+S**

The following prompt appears in the echo area.

```
I-Search:
```

2. Type the letter **d**.

The prompt in the echo area changes to

```
I-Search: d
```

The cursor moves to the first occurrence of "d" after its current position.

3. Type the letter **e**.

The prompt in the echo area changes to

```
I-Search: de
```

The cursor moves to the first occurrence of "de".

4. Type the letter **f**.

The prompt in the echo area changes to

```
I-Search: def
```

The cursor moves to the first occurrence of “def”.

This continues until you stop typing, or until the Editor fails to find the string you have typed in the current buffer. If at any point this does occur, the prompt in the echo area changes to reflect this. For instance, if your file contains the word “defun” but no word beginning “defm”, the prompt changes to

```
Failing I-Search: defm
```

as soon as you type **m**.

12.12.3 Replacing text

You can search for text and replace it with other text using the **Edit > Replace...** menu item. Type a string to search for and a string to replace it with in the Replace dialog that appears, and click **Find Next**. The cursor is placed immediately after the next occurrence in the current buffer of the string you specified. To replace this occurrence and locate the next one, click **Replace**. To leave this occurrence as it is and locate the next one, click **Find Next**. Note that this type of searching is *not* incremental.

For instance, assume you wanted to replace every occurrence of “equal” to “equalp”.

1. Choose **Edit > Replace...**.

   The Replace dialog appears.

2. Type **equal** in the **Find what** box:
3. Type **equalp** in the **Replace with** box and click **Find Next**.

The search will stop at every occurrence of “equal” after the current cursor position:

- If you want to replace this occurrence, click **Replace.**
- If you do not want to replace this occurrence, click **Find Next**.
- If you want to replace this occurrences and all later occurrences, click **Replace All**.
- If you want to abandon the operation altogether, click **Cancel**.

Note: Both **Edit > Find...** and **Edit > Replace...** start searching from the current position in the buffer. When the end of the buffer is reached, you are asked whether to start

again at the beginning. To start from the top of the buffer initially, check the **From Top** option before searching.

12.13 Using Lisp-specific commands

One of the main benefits of using the built-in editor is the large number of keyboard and menu commands available which can work directly on Lisp code. As well as editing facilities which work intelligently in a buffer containing Lisp code, there are easily-accessible commands which load, evaluate or compile, and run your code in any part of a buffer.

In addition, a high degree of integration exists between other Common LispWorks tools and the Editor. This allows you, for example, to find the source code definition of an object being examined in a browser, to set breakpoints in your code, or to flag symbols in editor buffers for specific actions, such as tracing or lambda list printing.

This section provides an introduction to the Lisp-specific facilities that are available using menu commands. For a full description of the extended editor commands, please refer to the *LispWorks Editor User Guide*.

All of the commands described below are available in the Editor's **Buffers**, **Definitions**, and **Expression** menus. They operate on the current buffers, definitions, or expression, the choice of which is affected by the current view.

12.13.1 Lisp mode

Some aspects of the LispWorks editor behave differently depending on which "mode" the buffer is using (see the *LispWorks Editor User Guide* for information about editor modes). These include syntax coloring and parenthesis matching. which operate only in Lisp mode and are described in "Syntax styles" on page 162. Also, certain commands such as those for indentation operate specially in Lisp mode.

To make a new buffer suitable for Lisp code, you can use the **`New Buffer`** command or the **File > New** menu item, both of which start the buffer in Lisp mode.

If your Lisp source files are saved with an extension **`.lisp`** or **`.lsp`**, then the editor will automatically open them in a Lisp mode buffer.

12.13.2 Current buffers, definitions and expression

In the Text view, the current buffer is the currently visible buffer, and the Buffers menu acts on this. The current expression is the symbol over which the cursor is positioned, or the one immediately before the cursor if it is not on a symbol. The current definition is the definition in which that current symbol occurs. For example:

```
(defun test ()
  (test2))
```

In the function shown above, if the cursor were placed on the letter "e" of **`test2`**, the current expression would be the symbol **`test2`**, and the current definition would be **`test`**.

In the Buffers view, the current buffer(s) are all the selected buffers. The **Definitions** and **Expression** menus are not available.

In the Definitions, Changed Definitions and Find Definitions views, the current definitions are all the selected definitions. The **Buffers** and **Expression** menus are not available.

In each view, the **Buffers**, **Definitions** and **Expression** menu commands act on the current buffer(s), definition(s) or expression.

12.13.3 Evaluating code

When you are editing Lisp code, you may want to evaluate part or all of the buffer in order to test the code. The easiest way to do this is using menu commands, although there are keyboard commands which allow you to evaluate Lisp in the Editor as well.

There are three menu commands which allow you to evaluate Lisp in the current buffer.

Choose **Buffers > Evaluate** to evaluate all the code in the current buffer. If you are in the buffers view, then this command evaluates the code in all the selected buffers.

Choose **Expression > Evaluate Region** to evaluate the Lisp code in the current region. You must make sure you have marked a region before choosing this command; see "Marking the region" on page 171. Whether you use the mouse or keyboard commands to mark a region does not matter. If you have a few Lisp forms that you want to evaluate, but do not want to evaluate the whole buffer, you should use this command.

Choose **Definitions > Evaluate** or click in the toolbar to evaluate the current definition. In the text view this is a little like evaluating the marked region, except that only the current definition is evaluated, whereas working with a marked region lets you evaluate several. This command is useful if you have a single function in the current buffer which you want to test without taking the time to evaluate the whole buffer or mark a region.

In the various definitions views, this command evaluates the code for all the selected definitions.

To load the file associated with the current buffer, choose **File > Load**. To load multiple files associated with buffers, select them in the buffers view and choose **File > Load**. If there is not a current buffer, the menu command **File > Load...** is available, which prompts for a file to load.

12.13.4 Compiling code

You can also compile Lisp code in an editor buffer in much the same way that you can evaluate it. Code can be compiled in memory or to a file.

12.13.4.1 Compiling in memory

Choose **Buffers > Compile** or click in the toolbar to compile all the code in the current buffer.

Choose **Expression > Compile Region** to compile the Lisp code in the current region.

Choose **Definitions > Compile** or click in the toolbar to compile the current definition.

During compilation, the Editor tool temporarily displays compiler output in the Output tab. Once compilation has finished, you can press **`Space`** to display the current buffer once again.

Additionally, if any conditions were signalled during the compilation, you can view these in the Compilation Conditions Browser by pressing **`Return`**. You can also locate the source code that generated a message via the context menu, as described in “Interactive compilation messages” on page 307.

You can review the output at any time by clicking the Output tab of the Editor.

12.13.4.2 Compiling to a file

To compile the file associated with the current buffer, choose **File > Compile**. To compile multiple files associated with buffers, select them in the buffers view and choose **File > Compile**. If there is not a current buffer, the menu command **File > Compile...** is available, which prompts for a file to compile.

Note: this command calls the Common Lisp function **`compile-file`**; it creates the fasl file but does not load it. You can use **File > Load** to later load the fasl.

To compile a file (or files) and load the resulting fasl file(s) with a single command, choose **File > Compile and Load**. If there is not a current buffer, the menu command **File > Compile and Load...** is available.

12.13.5 Breakpoints

A breakpoint causes execution of Lisp code to stop when it is reached, and the LispWorks IDE displays the stack and the source code in a Stepper Tool. See “Breakpoints” on page 350 for information about using breakpoints with the Stepper Tool.

A breakpoint can be at the start, function call or return point of a form.

12.13.5.1 Setting breakpoints

To set a breakpoint, for example at the call to **+** in one of your functions:

1. Open the file containing the call in an Editor tool.
2. Ensure the definition is indented. You can use the Lisp mode command **`Indent Form`** (**`Meta+Ctrl+Q`** in Emacs emulation).
3. Ensure the definition is compiled.
4. Position the cursor on the symbol **+**.
5. Choose the menu command **Expression > Toggle Breakpoint**, or click in the Editor toolbar, or run the editor command **`Toggle Breakpoint`**. The symbol **+** is highlighted red indicating that a breakpoint is set.

When the breakpoint is reached, a Stepper tool is invoked, allowing you to step through the code, add further breakpoints, and so on. See “The Stepper” on page 339 for more information about the Stepper tool.

12.13.5.2 Editing breakpoints

To edit the Conditional or Printing properties of a breakpoint, choose the menu command **Expression > Edit Breakpoints** and proceed as described in “Editing breakpoints” on page 354.

To visit the source code where a breakpoint was set, choose the menu command **Expression > Edit Breakpoints**, select a breakpoint and press the **Goto Source** button. This cancels the dialog and then displays the source containing the breakpoint.

12.13.5.3 Removing breakpoints

To remove a breakpoint under the cursor, click in the toolbar. Equivalently choose the menu command **Expression > Toggle Breakpoint** or run the editor command **`Toggle Breakpoint`**.

Where you wish to remove one or more breakpoints without finding them in the source, choose **Expression > Edit Breakpoints**, select a breakpoint or breakpoints in the Breakpoints list, and click **Remove**.

12.13.5.4 Reloading code with breakpoints

A message like this:

```
Retain 1 breakpoint from loaded file...
```

means that a breakpoint is set in a buffer while you have loaded that buffer's underlying file from disk, for example by menu commands **File > Load** or **File > Compile And Load**. Loading the file re-evaluates all of its forms, but the IDE does not have a way to reset the breakpoints in these forms automatically. Therefore it asks you what to do.

Answer **Yes** to add breakpoints to the newly loaded definitions. Answer **No** to remove the breakpoints.

12.13.6 Tracing symbols and functions

A wide variety of tracing operations are available in the **Buffers**, **Definitions** and **Expression** menus. The scope of each operation depends on which menu the command is chosen from.

Choose **Trace** from either the **Buffers**, **Definitions** or **Expression** menus to display a menu of trace commands that you can apply to the current region or expression, or the

currently selected buffers or definitions, as appropriate. Note that you can select several items in the buffers and definitions views.

See “Tracing symbols from tools” on page 40 for full details of the tracing facilities available in the Editor.

12.13.7 Packages

It is important to understand how the current package (that is, the value of the Common Lisp variable **`*package*`**) is determined when running Lisp operations such as evaluation or compilation commands in a buffer. Usually it is obvious: most Lisp source files have a single **`in-package`** form. The Editor uses the specified package as the current package when you evaluate or compile code in that buffer, or perform some other operation that depends on the current package.

However it is possible for a source file to contain multiple **`in-package`** forms, or none at all. In this case, the Editor uses a suitable binding for the current package depending on the location in the buffer, as described below. This means that you do not have to worry about setting the package explicitly before evaluating part of a buffer, and that operations within a buffer use the expected current package.

12.13.7.1 The primary package

Each buffer has a package associated with it, known as the primary package. This is set when the buffer is created, and is displayed in the message area at the bottom of the Editor window. The primary package provides a default, used when the current package cannot be determined by other means.

If the buffer is created by opening a file containing an **`in-package`** form, that package is the primary package. If there are multiple **`in-package`** forms, the primary package is taken from the first of these forms. If there is no **`in-package`** form, the primary package is **`CL-USER`**.

You can set the primary package if needed with the editor command **`Set Buffer Package`**. See the *LispWorks Editor User Guide* for details.

12.13.7.2 The current package for Lisp operations

When evaluating or compiling an entire buffer, the Editor uses **`in-package`** forms as they appear in the code. For any code that precedes the first **`in-package`** form, or

when there is no **in-package** form, the code is evaluated or compiled in the primary package.

When evaluating or compiling a region of the current buffer (as opposed to all of it), the Editor uses **in-package** forms as they appear in the region. For any code that precedes the first **in-package** form of the region, or when there is no **in-package** form in the region at all, the Editor searches for the previous **in-package** form in the buffer. If this is found, it determines the current package, otherwise the primary pacakge is used.

When evaluating or compiling a definition, and for operations such as symbol completion at the cursor point, the Editor searches for the previous **in-package** form in the buffer. If this is found, it determines the current package, otherwise the primary pacakge is used.

12.13.8 Indentation of forms

The Editor provides facilities for indenting your code to help you see its structure. These facilities are available only in Lisp mode. The EMACS key **Alt+Ctrl+Q** indents the current Lisp form, and the **Tab** key indents a single line.

You can customize Lisp mode indentation by using the **Defindent** command, see the *LispWorks Editor User Guide* for details.

See "Lisp mode" on page 178 for more information about Lisp mode.

12.13.9 Other facilities

A number of other Lisp-specific facilities are available using the menus in the Editor.

If the current buffer is associated with a file that is part of a system as defined by **defsystem**, choose **File > Browse Parent System** to browse the system it is part of in the System Browser. See Chapter 26, "The System Browser" for more information about this tool.

Choose **Definitions > Undefine...** to remove the current definitions from your Lisp image. Similarly, choose **Buffers > Undefine...** to remove the definitions in the current buffer or selected buffers. By selecting items in the buffers view, or the various definitions views, you can control over the definitions which can be removed with one command. Both of these commands prompt you for confirmation with a second chance to modify the list of definitions to remove.

Choose **Definitions > Generic Function** to describe the current definition in a Generic Function Browser. See Chapter 14, “The Generic Function Browser” for more details.

Standard action commands can be found on the **Expression** menu, allowing you to perform a number of operations on the current expression. See “Performing operations on selected objects” on page 35 for full details.

Choose **Expression > Arguments** to print the lambda list of the current expression in the echo area, if it is a function, generic function or method. This is the same as using the EMACS key command **`Alt+=`**, except that the current expression is automatically used.

Choose **Expression > Value** to display the value of the current expression in the echo area.

Choose **Expression > Macroexpand** or click in the toolbar to macroexpand the current form. The macroexpansion is printed in the Output tab, in the same way that compilation output is shown. Note how an **`in-package`** form containing the current package is printed with the macroexpansion, meaning that you can preform a further macroexpansion. Press **Space** when the cursor is at the end of the output window to return to the Text tab.

Choose **Expression > Walk** to recursively macroexpand the current form.

12.14 Help with editing

Two help commands are available which are specific to the Editor and any tools which use editor windows.

Choose **Help > Editing > Key to Command** and type a key sequence to display a description of the function it is bound to, if any.

Choose **Help > Editing > Command to Key** and supply an editor command name to see the key sequence it is bound to, if any.

13

The Function Call Browser

13.1 Introduction

The Function Call Browser gives you a way to view a user-defined function in the Lisp image together with the functions that call it or the functions it calls.

It has three views.

- The Called By view allows you to examine a graph of the functions which call the function being browsed. This is the default view.
- The Calls Into view allows you to examine a graph of the functions which are called by the function being browsed.
- The Text view lets you see immediate callers and callees of the browsed function using lists rather than a graph.

To create a Function Call Browser, choose **Works > Tools > Function Call Browser** or click in the Podium. Alternatively, select a function in another tool, and choose **Function Calls** from the appropriate actions menu to browse the selected function in the Function Call Browser. Finally, in an editor executing **`Alt+X List Callers`** or **`Alt+X List Callees`** calls up a Function Call Browser on the current function.

Note: the cross references between function calls are generated by the compiler, hence you can use the Function Call Browser only for compiled code. Moreover, the com-

piler setting to generate cross references must be on when you compile your code. Switch it on by evaluating

```
(toggle-source-debugging t)
```

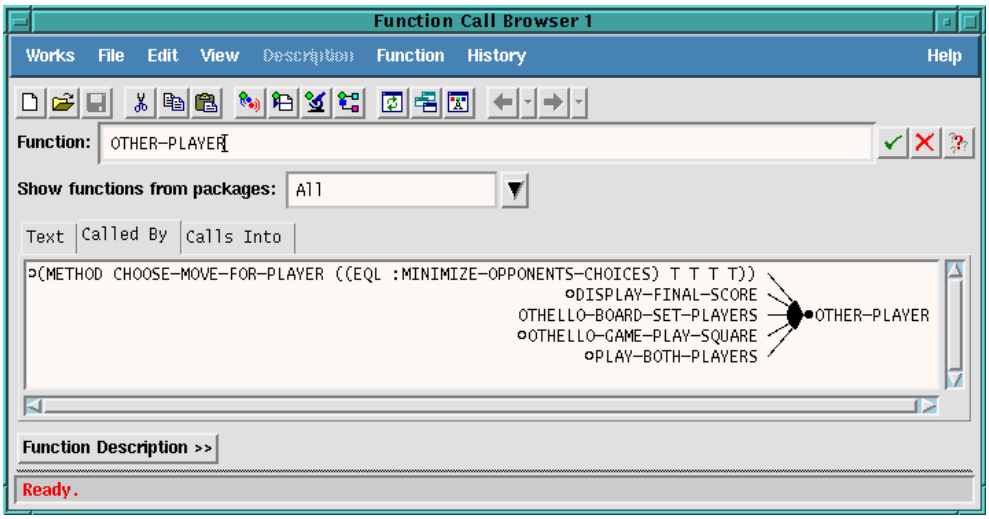
When cross referencing is on, this line appears in the output of the compiler:

```
;;; Cross referencing is on
```

13.2 Examining functions using the graph views

There are two graph views in the Function Call Browser. The Called By view is the default view. The Function Call Browser appears as in Figure 13.1.

Figure 13.1 Viewing functions using the “Called By” view



In this view, the Function Call Browser has five areas.

13.2.1 Function area

The Function area displays the name of the function being examined, and here you can enter the name of another function to examine. You can use completion to reduce typing. This allows you to select from a list of all functions in the current package whose names begin with the partial input you have entered. Invoke completion by **Up**, **Down**

or click the button. See "Completion" on page 46 for detailed instructions. When you have entered the complete function name, click to confirm your choice

13.2.2 Show functions control

The popup list **Show functions from packages** allows you to restrict the functions displayed based on their package. It affects the display in all views. Below, *the current package* means the **symbol-package** of the function currently being examined in the Function Call Browser. The options are:

All Display all the functions known to the compiler.

Current and Used
Display only those functions in the current package or packages on the package use list of the current package.

Current and Standard
Display only those functions in the current package or the standard packages **COMMON-LISP**, **HCL** and **LISPWORKS**.

Current Display only functions in the current package.

13.2.3 Graph area

A graph of all the callers of the function is displayed in a graph in the *Called By* view. The graph area of the *Calls Into* view is similar, but the graph displayed is of the functions called by the function being browsed.

Note that if source level debugging is off, or the function was not compiled, there is no information to display here. To turn on source level debugging, call

```
(toggle-source-debugging t)
```

The generic facilities available to all graph views in the LispWorks IDE are available here; see Chapter 5, "Manipulating Graphs" for details.

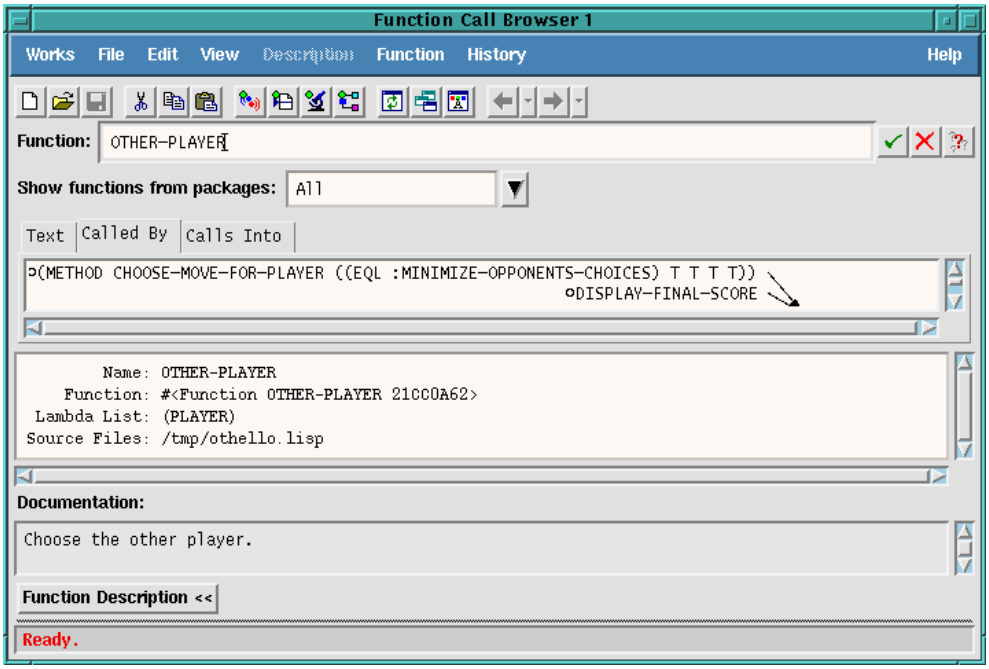
13.2.4 Echo area

The echo area of the Function Call Browser is similar to the echo area of the podium. It displays messages concerning the Function Call Browser.

13.2.5 The function description button

Clicking on **Function Description >>** changes the view of the Function Call Browser to include more information on the function being browsed. The browser appears as in Figure 13.2

Figure 13.2 The Function Call Browser in function description mode



Two further panes appear. Note that the function description button has now changed to **Function Description <<** and that clicking on it restores the view of the Function Call Browser

The extra panes are a function description area, and a documentation area.

13.2.5.1 Function description area

The Function Description area gives a description of the function selected in the main-area, or, if nothing is selected, the current function (as displayed in the Function area). The following items of information are displayed:

Name	The name of the function.
Function	The function object.
Lambda List	The lambda list of the function.
Source Files	The source file in which the function is defined, if any.

You can operate on any of the items in this area using the commands in the **Description** menu, which is also available as the context menu. This contains the standard actions described in “Performing operations on selected objects” on page 35.

13.2.5.2 Documentation area

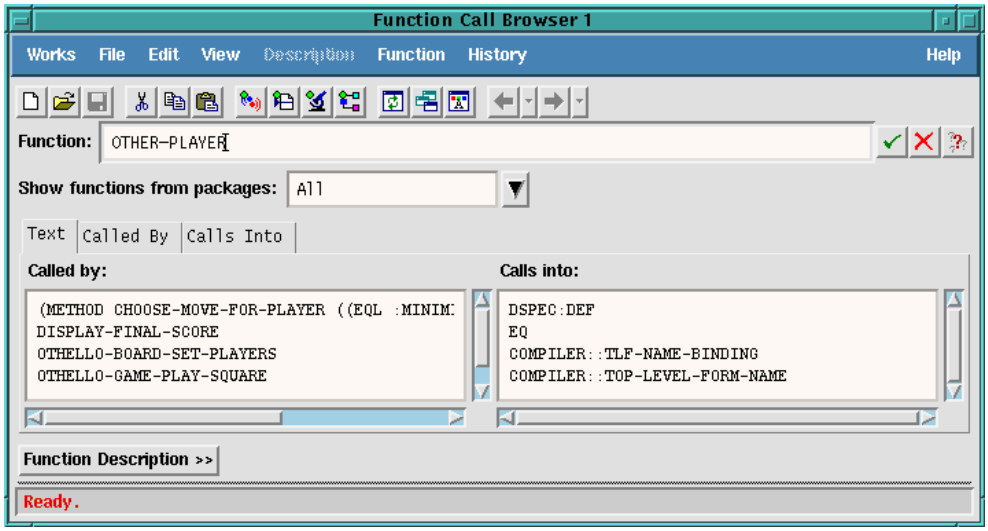
The Documentation area shows the documentation for the function selected in the main area as returned by the Common Lisp function `documentation`. If no function is selected, the documentation for the current function is shown.

13.3 Examining functions using the text view

Click on the Text tab to see a textual display of the callees and callers of a function. This view has the advantage that both callees and callers can be seen simultaneously. It is very similar to the text view in the Class Browser, as described in “Examining

other classes” on page 77. When in the text view, the Function Call Browser appears as shown in Figure 13.3.

Figure 13.3 Viewing functions using the text view



The function area, show functions from packages area, function description area and echo area are as in the graph views.

13.3.1 Called By area

The Called By area lists those functions which the current function calls.

To make any function in this list be the current function, double-click on it.

13.3.2 Calls Into area

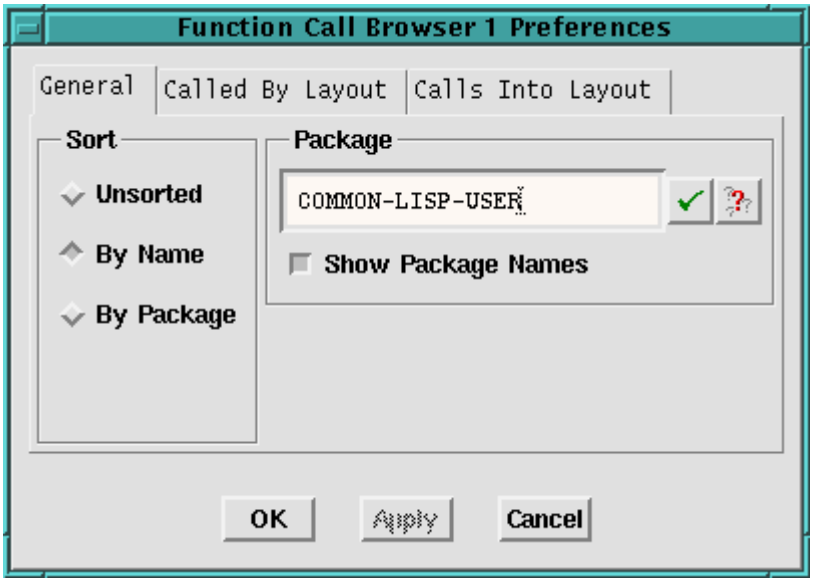
The Calls Into area lists those functions which call the current function.

To make any function in this list be the current function, double-click on it.

13.4 Configuring the function call browser

The Function Call Browser can be configured using the preferences dialog. Select **Works > Tools > Preferences...** or click in the Function Call Browser to display the dialog, which is shown in Figure 13.4

Figure 13.4 The function call preferences dialog.



13.4.1 Sorting entries

The functions displayed in each tab of the Function Call Browser can be sorted in a number of ways.

Choose **By Name** to sort entries according to the function name. This is the default setting.

Choose **By Package** to sort functions according to their package.

Choose **Unsorted** to leave functions unsorted.

13.4.2 Displaying package information

As with other tools, you can configure the way package names are displayed in the Function Call Browser.

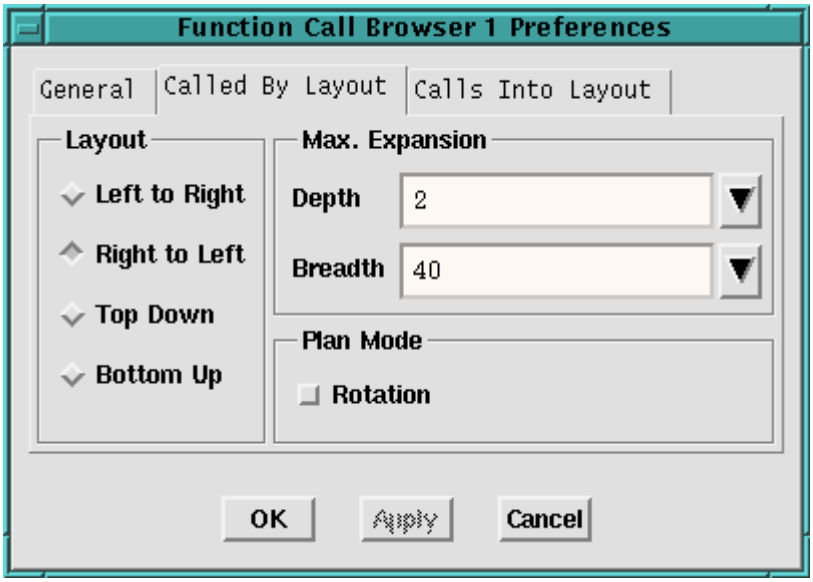
Choose **Show Package Names** to turn on and off the display of package names in the Graph, Called By, Calls Into and Description areas.

See “Displaying packages” on page 32 for more information about using **Show Package Names.**

13.5 Configuring graph displays

The preferences dialog can also be used to configure how the Function Call Browser displays graphical information in the Called By and Calls Into views. Click on the Called By Layout or Calls Into Layout tab in the preferences dialog. Both views perform the same operations on the relevant Function Call Browser view.

Figure 13.5 A layout view in the preferences dialog



13.5.1 Graph layout settings

The layout radio buttons are used to set the direction in which the graph is displayed. The default setting is **Left to Right**.

13.5.2 Graph expansion settings

The Max. Expansion settings determine how much of the graph to display. The default depth value is 2—this ensures that only functions that directly call (or are directly called by) are shown in the graph. If this value were set to 3, for example, then functions that call a function that calls the function being browsed would also be displayed.

The breadth value has a default value of 40, and sets how many functions are displayed at each level of the graph.

13.5.3 Plan mode settings

The **Rotation** checkbox determines whether the graph layout can be rotated when in plan mode.By default it is unchecked.

You can enter plan mode when displaying a graph by selecting **Enter Plan Mode** from the context menu. If rotation is enabled and the plan is smaller than the graph, you can rotate the plan by holding down the **`Shift`** key and moving the mouse left or right.

13.6 Performing operations on functions

A number of operations can be performed on functions selected in the Text area (when in the text view) or in the Called By or Calls Into areas, or on the current function (when there are no functions selected elsewhere).

The **Function** menu gives you access to the standard actions described in “Performing operations on selected objects” on page 35.

The **Function > Trace** submenu gives you the ability to trace and untrace the functions selected in the Text, Called By and Calls Into views.

14

The Generic Function Browser

The Generic Function Browser allows you to examine the generic functions in the Lisp image, together with any methods that have been defined on them. It has two views which allow you to browse different types of information:

- The methods view, which shows you a description of the generic function and the methods defined on it. This is the default view.
- The method combinations view, which lets you examine the list of method combinations for any generic function.

To create a Generic Function Browser, choose **Works > Tools > Generic Function Browser** or click in the Podium.

Other ways to create a Generic Function Browser are:

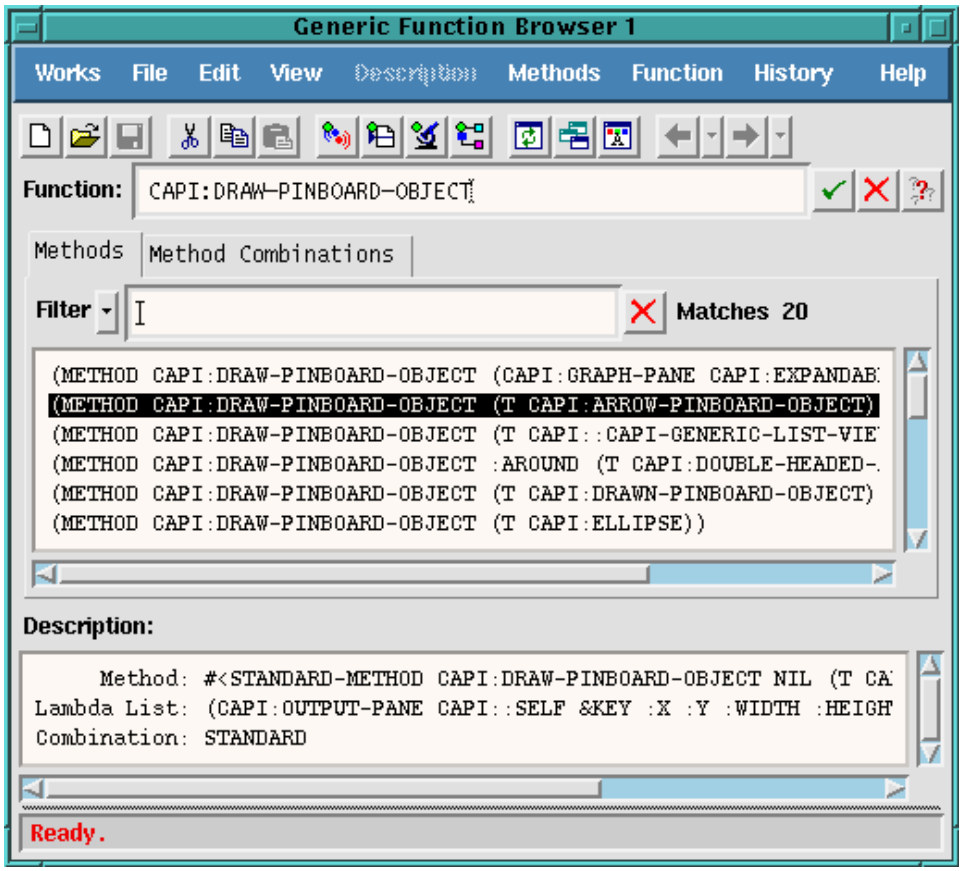
- If the current object in a tool is a generic function or method, choose the **Generic Function** standard action command from the appropriate menu
- Use the editor command **`Alt+X Describe Generic Function`**
- If there is a method on the debugger stack, you can display the Method Combination via the **Frame** menu of a Debugger tool

14.1 Examining information about methods

When the Generic Function Browser is first displayed, the default view is the methods view. You can also choose it explicitly by clicking on the Methods tab of the Generic Function Browser.

The methods view is shown in Figure 14.1 below.

Figure 14.1 Generic function browser



The methods view has four main sections, described below.

14.1.1 Function area

The Function box shows the name of the generic function you are examining. To browse a generic function, enter its name into the Function box, or paste it in from another tool in one of two ways:

- Choose **Edit > Copy** or the standard action command **Copy** in another tool to copy the generic function to the clipboard, then choose **Edit > Paste** in the Generic Function Browser to transfer the generic function in.
- Choose the standard action command **Generic Function** in the other tool to display the generic function in the Generic Function Browser in one action.

When entering the name of a function, you can use completion to reduce typing. This allows you to select from a list of all generic functions whose names are accessible in the current package and begin with the partial input you have entered. Invoke completion by pressing `Up` or `Down`, or by clicking the button. The methods are listed immediately. See “Completion” on page 46 for more information about completion. If you enter the generic function name directly without using completion, click to confirm the name.

Note: You can use **Edit > Paste** to paste in a generic function, even if the Common LispWorks clipboard currently contains the string representation of the function, rather than the function itself. This lets you copy in generic functions from other applications, as well as from the environment. See “Using the clipboard” on page 28 for a complete description of the way the Common LispWorks clipboard operates, and how it interacts with the UNIX clipboard.

You can operate on the current generic function using the commands in the Generic Function Browser’s **Function** menu. See “Performing operations on the current function or selected methods” on page 200 for details.

14.1.2 Filter area

The Filter lets you restrict the list of methods displayed. See “Filtering information” on page 42 for details about how to use the Filter area.

14.1.3 Methods list

This area displays the methods defined on the generic function.

- Selecting a method in this list displays its description in the Description list.
- Double-clicking on a method displays its source code definition in the editor, if it is available.

The number of items listed in the list of methods is printed in the Matches box.

You can operate on any number of selected methods in this area using the commands in the Generic Function Browser's **Methods** menu. See Section 14.1.5 on page 200 for details.

14.1.4 Description list

The Description list shows a description of the method selected in the list of methods, or of the generic function itself if no method is selected.

The following information is listed:

Method	The method object that is selected in the list of methods.
Lambda List	The lambda list of the generic function.
Combination	The class of method combination for the generic function.

To operate on any of the items displayed in this area, select them and choose a command from the **Description** menu. This menu contains the standard action commands described in "Performing operations on selected objects" on page 35. You can operate on more than one item at once by making a multiple selection in this area.

14.1.5 Performing operations on the current function or selected methods

You can use the **Function** and **Methods** menus to access commands that operate on the current generic function or the selected methods. These commands are similar to commands available in other tools, and so you should find them familiar.

The following commands are available from either the **Function** or **Methods** menus:

- The standard action commands let you perform a number of operations on the selected methods or the current function. For details on the commands available, see “Performing operations on selected objects” on page 35.
- Choose **Undefine...** to undefine the current generic function or the selected methods so that they are no longer available in the Lisp image. Choosing **Undefine...** on a method undefines the method function and removes it from the methods of the generic function. However, the generic function can still be called with its different method selection.
- The **Trace** submenu gives you the ability to trace and untrace the current generic function or the selected methods. See “Tracing symbols from tools” on page 40 for details about the commands available in this submenu.

14.2 Examining information about combined methods

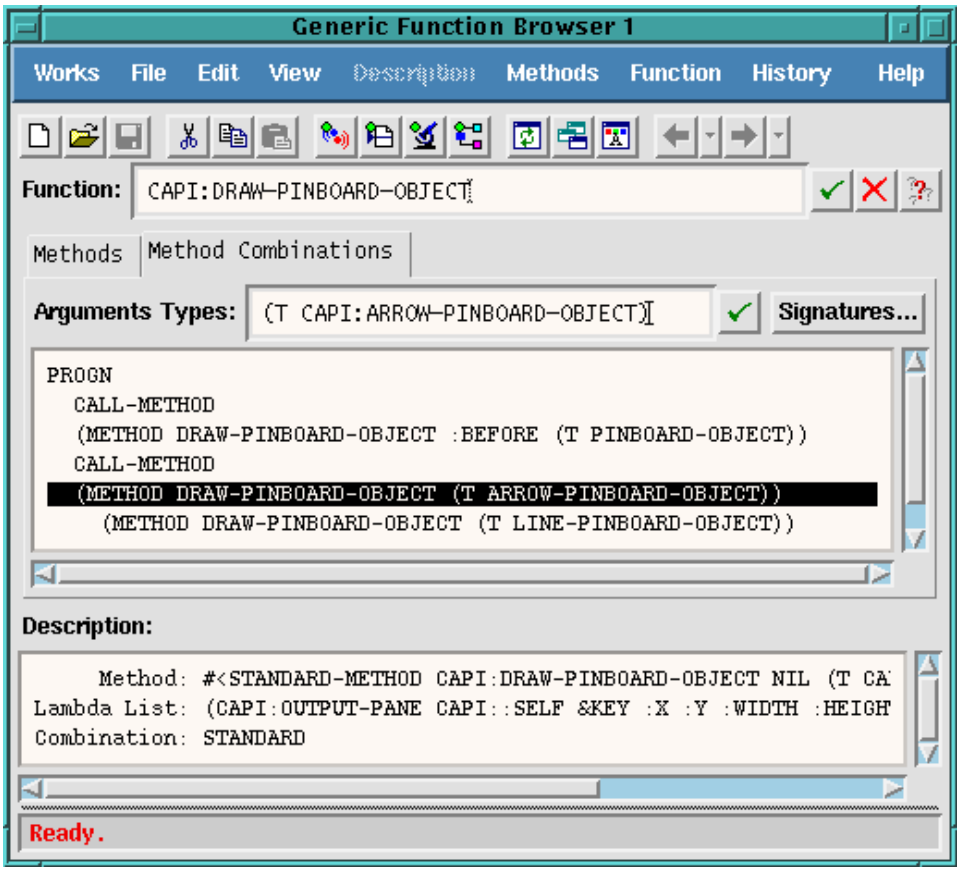
The method combinations view lets you examine information about the combined methods of the current generic function. You supply a signature and Generic Function Browser displays the combined methods of the generic function together with the arguments that match that method combination point.

Method combinations show you the calling order of methods. They use the class precedence lists of the classes on which the methods of a generic function operate. Being able to view these combinations gives you a simple way of seeing how before, after, and around methods are used in a particular generic function.

You can display this view by clicking the Method Combinations tab of a Generic Function Browser, or from the Debugger tool by choosing **Frame > Method Combina-**

tion in a frame containing a standard method. The method combinations view is shown in Figure 14.2 below.

Figure 14.2 Generic function browser displaying method combinations



The method combinations view has a number of main sections, described below.

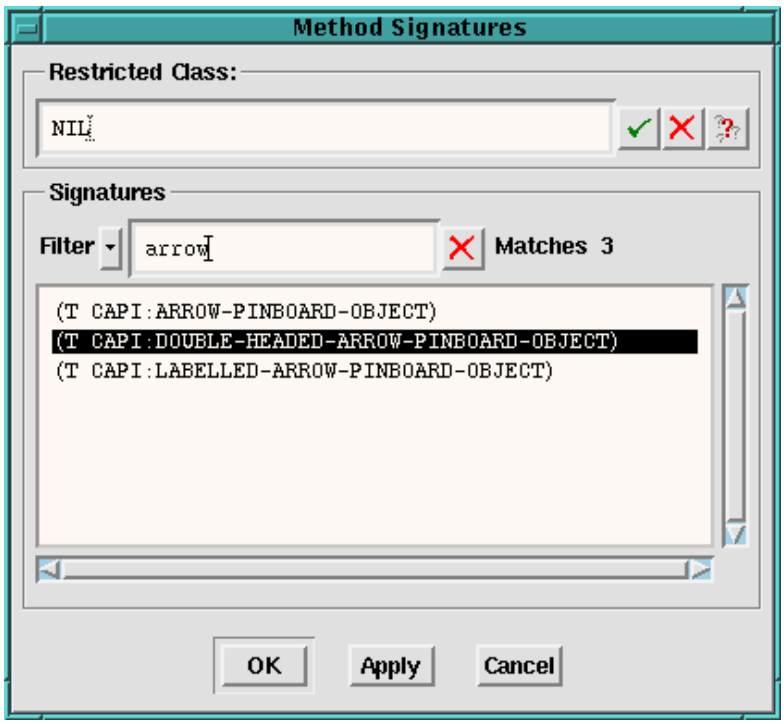
14.2.1 Function box

As with the methods view, the name of the generic function being browsed is shown here. See “Function area” on page 199 for details.

14.2.2 Signatures button

Click **Signatures...** to display the Method Signatures dialog shown in Figure 14.3. This dialog lists the signatures for the methods defined on the current generic function. The signature of a method shows the types of the arguments.

Figure 14.3 Method Signatures dialog



To list the method combinations of any defined method in the Generic Function Browser, select its signature from the list in the Signatures panel of the Method Signatures dialog and click **OK**.

You can restrict the signatures displayed using Filter box in the usual way.

You can also restrict the display with the Restricted Class box. See “Restricting displayed signatures by class” on page 204 for details.

14.2.3 Arguments types box

The Arguments types box is used to specify a signature, in order to see the method combinations. You can specify a signature by either:

- Choosing a signature using the Method Signatures dialog, as described in “Signatures button” on page 203.
- Typing the signature list in directly and clicking .

The method combinations for the relevant method are displayed in the list of method combinations.

14.2.4 List of method combinations

The main list in the method combinations view shows method combinations for the signature specified in the Arguments Types box.

- Selecting any method in the list displays its description in the Description list.
- Double-clicking on any method in the list displays its source code definition in the editor, if it is available.

You can operate on any number of selected methods in this area using the commands in the **Methods** menu. See “Performing operations on the current function or selected methods” on page 200 for details.

14.2.5 Description list

The Description list displays a description of any method selected in the list of method combinations. The same items of information are shown as in the methods view; see “Description list” on page 200.

To operate on any of the items displayed in this area, select them and choose a command from the **Description** menu. This menu contains the standard commands described in “Performing operations on selected objects” on page 35. You can operate on more than one item at once by making a multiple selection.

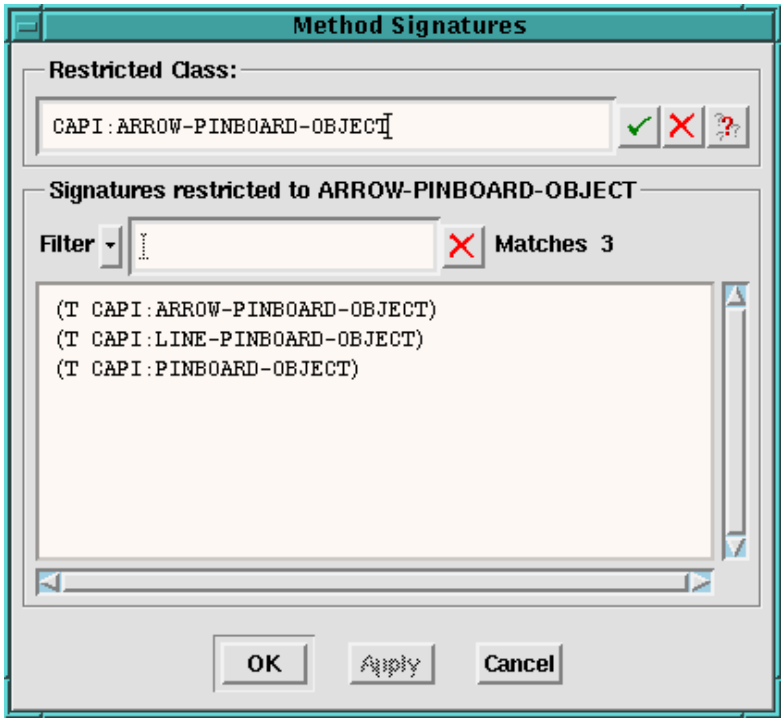
14.2.6 Restricting displayed signatures by class

The Method Signatures dialog was introducted in “Signatures button” on page 203. You can display this dialog by clicking **Signatures...** in the Generic Function Browser.

By default, the Method Signatures dialog displays the signatures of all methods defined on the generic function. When there are many methods, or when the distinction between different classes is not clear, this can be confusing.

To simplify the display, you can restrict the signatures displayed to a chosen class and its superclasses. To do this, enter the name of the chosen class into the **Restricted Class** box. You can click which allows you to select from a list of all class names which begin with the partial input you have entered. See “Completion” on page 46 for detailed instructions. As with similar text input panes in the IDE, click to confirm your choice, to cancel the current setting.

Figure 14.4 Restricting the signatures by class



Once you have made a choice, only those signatures that contain the specified class or one of its superclasses are listed in the Signatures panel of the dialog. This simplified display is useful when there are a large number of complicated signatures.

Be aware of the difference between this approach and the use of the Argument box in the Signatures panel. Restricting signatures confines the signatures offered in the dialog by means of the class of the signatures.

Click to display the signatures for all methods defined once again.

14.3 Configuring the Generic Function Browser

As with other Common LispWorks tools, choose **Works > Tools > Preferences...** or click to display the Generic Function Browser Preferences dialog.

Using the options in the Sort panel, you can sort the items in the Generic Function Browser as you can in many of the other Common LispWorks tools.

Unsorted — Displays items in the order they are defined in.

By Method Qualifier — Sorts items by the CLOS qualifier of the method. This groups together any `:before`, `:after`, and `:around` methods.

By Name — Sorts items alphabetically by name. This is the default setting.

By Package — Sorts items alphabetically by package name.

For more information on sorting items, see “Sorting items in views” on page 38.

You can also set the process package of the Generic Function Browser, and choose to hide package names in the display, using the Package box. See “Displaying packages” on page 32 for full details.

You can also control whether the Generic Function Browser displays the history toolbar by the option **Show Toolbar**, as described in “Toolbar configurations” on page 24.

15

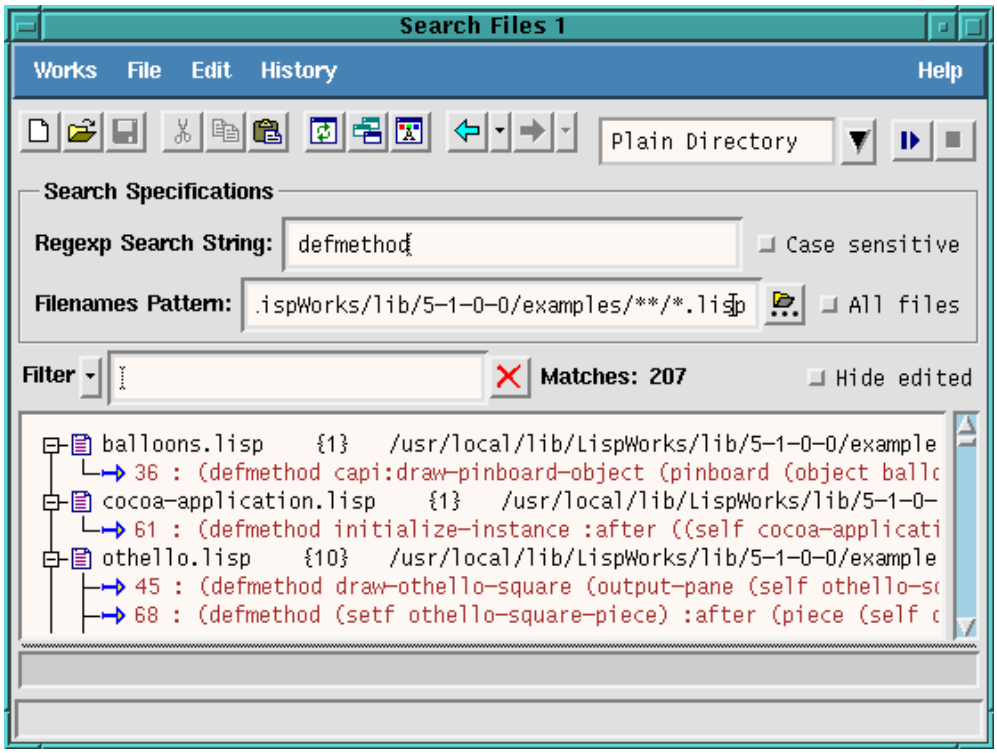
The Search Files tool

15.1 Introduction

The Search Files tool gives you a convenient way of performing searches on directories, individual files or systems. You can create a Search Files tool by choosing **Works > Tools > Search Files** or clicking in the Podium. You can also start context-dependent searches, for example by choosing **Edit > Search Files...** or **Systems > Search Files...**, or from editor command such as **`Meta+X Search Files`**.

Out of necessity, this chapter makes some references to other tools in the environment which you may not yet be familiar with. However, this chapter does not assume any prior knowledge of these tools.

Figure 15.1 The Search Files tool



The Search Files tool has the following areas:

- The toolbar contains a dropdown list that chooses the kind of search to perform (**Plain Directory** was used in the screenshot above). There are also buttons to start and stop a search.
- The **Search Specifications** area lets you specify where to search and what to search for. This area is filled in or partly filled in automatically when you start a context-dependent search. You can also enter suitable values directly, or modify the existing values.
- The filter area lets you restrict the search results displayed in the main area.

- The main area displays the results of the last search.

15.2 Performing searches

You can use the Search Files tool in two different ways.

- You can enter details of where to search and what to search for directly into the tool and click the button. This is described in more detail in “Entering Search Specifications directly” on page 209.
- You can use an Editor command or menu command that starts a context-dependent search. This is described in more detail in “Using context-dependent searches” on page 214.

Some kinds of the search use a regular expression (regexp). For details of the syntax of regular expressions see "Regular expression searching" in the *LispWorks Editor User Guide*.

While the tool is searching, you can examine the results but you cannot change the search specifications. To stop a search, click the button in the toolbar.

15.2.1 Entering Search Specifications directly

To enter the search specifications directly, decide which kind of search to perform from the dropdown list in the toolbar and then fill in the boxes in the **Search Specifications** area. The different search kinds are described below. You can also hide the search specifications by choosing **Hide Search Area** from the dropdown list in the toolbar.

15.2.1.1 Plain Directory searches

A **Plain Directory** search is used to search for a particular regexp in all files whose names match a particular pattern. Enter the regexp in the **Regexp Search String** box and enter a set of filename patterns in the **Filenames pattern** box. You can press **Up** or **Down** in the **Filenames pattern** box to complete physical directory components, as described in “Completion” on page 46.

The filename pattern should be a complete filename and can use the following syntax to make it match more than one file:

- Use **`*`** within the pattern to match any sequence of characters in a directory or file name.
- Use **`**`** within the the directory part of the pattern to match any number of subdirectories.

Here are some examples of filename patterns:

`*.*` Matches all files in the root directory.

`subdir/*.txt`
Matches all **`txt`** files in *root*`/subdir`.

`examples/**/*.lisp`
Matches all **`lisp`** files in *root*`/examples` and its subdirectories. This is similar to the pattern shown in Figure 15.1.

`**/*zork*/*.bmp`
Matches all **`bmp`** files in any directory under the root directory that contains **`zork`** in its name

See also the **Match flat file-namestring** option in "Search Parameters" on page 217 for additional information.

If a filename pattern is a directory then all files in that directory are searched.

Check **Case sensitive** to make the search match only the case of letters exactly as entered.

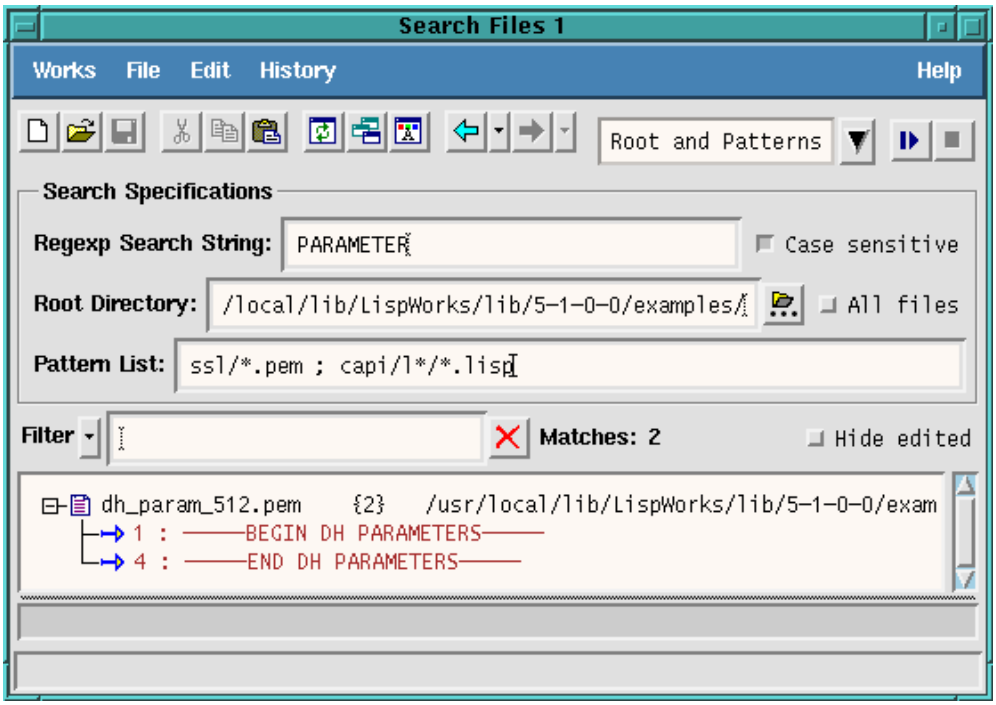
Check **All files** to ignore any list of **File Types** in the Preferences.

15.2.1.2 Root and Patterns searches

A **Root and Patterns** search is used to search for a particular regexp in all files whose names match one or more patterns within a directory. Enter the regexp in the **Regexp Search String** box, the starting directory in the **Root Directory** box, and a set of filename patterns in the **Pattern List** box.

You can press **Up** or **Down** in the **Root Directory** box to complete physical directory components, as described in "Completion" on page 46.

Figure 15.2 A Roots and Patterns search



You can search subdirectories by including directory components (including wild components) in the the **Pattern List** box.

Multiple filename patterns can be entered, separated by semicolons. Spaces before and after each pattern are ignored. Each filename pattern should be a complete filename and can use the following syntax to make it match multiple files:

- Use ***** within the pattern to match any sequence of characters in a directory or file name.
- Use ****** within the the directory part of the pattern to match any number of sub-directories.
- Use **{name1,name2,...}** to match any one of **name1**, **name2** and so on. Spaces before and after each name are ignored.

Here are some examples of pattern lists:

```
images/*.*  ; icons/*.*

{images,icons}/*.*
```

Both of these patterns match all files in the *root*`/images` and the *root*`/icons` directories.

```
**/{images,icons}/sunrise.{bmp,jpg,jpeg}

**/images/sunrise.{bmp,jpg,jpeg} ; **/icons/
sunrise.{bmp,jpg,jpeg}
```

Both of these patterns match all files with the name `sunrise.bmp`, `sunrise.jpg` or `sunrise.jpeg` in a directory named `icons` or `images`, anywhere in the root directory.

```
{maj,min}or-events/{*-name,date}/*.txt

major-events/{*-name,date}/.txt ; minor-events/{*-name,date}/.txt

{maj,min}or-events/date/*.txt ; {maj,min}or-events/*-name/*.txt
```

Each of these three patterns matches all `.txt` files which are in a directory `date` or a directory that ends with `-name` in the `major-events` or `minor-events` directories.

See also the **Match flat file-namestring** option in “Search Parameters” on page 217 for additional information.

If a filename pattern is a directory then all files in that directory are searched.

Check **Case sensitive** to make the search match only the case of letters exactly as entered, as illustrated above.

Check **All files** to ignore any list of **File Types** in the Preferences.

15.2.1.3 System Search

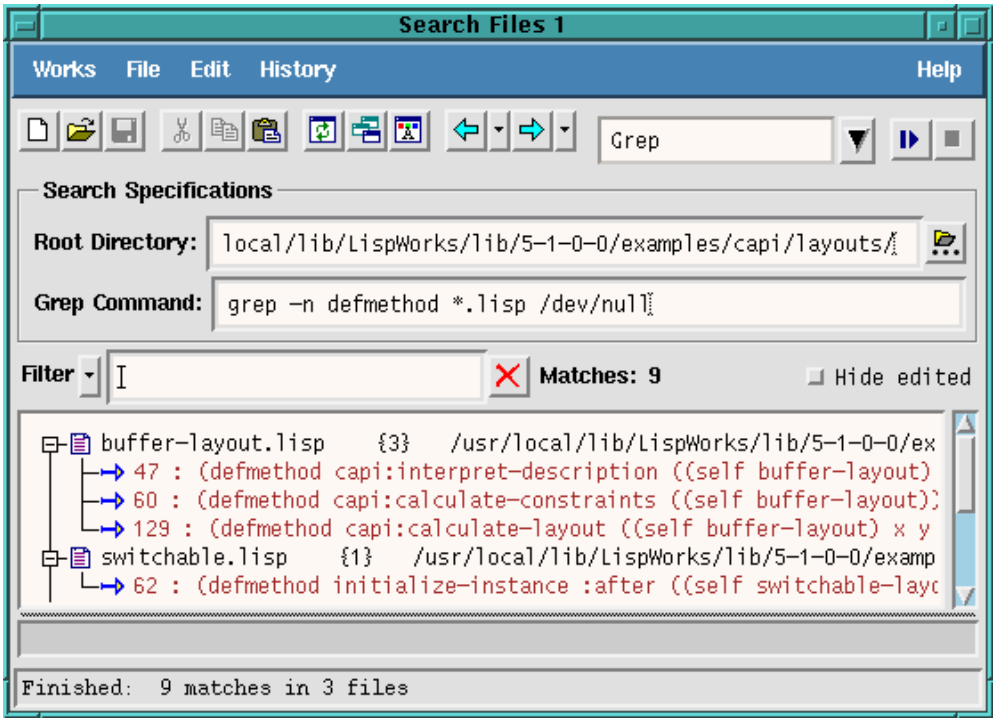
A **System Search** is used to search for a particular regexp in all the files referenced by a LispWorks `defsystem` definition. Enter the regexp in the **Regexp Search String** box and the system names in the **System Names** box. Multiple system names can be entered, separated by semicolons.

Check **Case sensitive** to make the search match only the case of letters exactly as entered.

15.2.1.4 Grep searches

A **Grep** search is used to run an external program to search files and show the results in the tool. Enter the working directory for the external program in the **Root Directory** box and the complete command line of the external program in the **Grep Command** box.

Figure 15.3 A Grep search



The external program is typically **grep**, but other programs can be used as long as they print the matched lines in this format:

```
filename:line-number line-text
```

When using **grep** you generally need to pass the **-n** option and the filename argument **/dev/null** to force it to print the file and line number in all cases. This is done automatically when you invoke the Search Files tool by the Editor command **Grep**.

15.2.2 Using context-dependent searches

Context dependent searches take some information from the current window and invoke the Search Files tool to perform the search. There are various Editor commands and menu commands that start a context-dependent search, as described below.

15.2.2.1 Context-dependent searches using Editor commands

`Search Files`

Prompts for a search string and directory pattern and then performs a **Plain Directory** or **Root and Patterns** search. If an existing Search Files tool is reused by this command and was last doing a **Root and Patterns** search, then the directory pattern is split to fill the boxes. Otherwise, a **Plain Directory** search is performed using the directory pattern. If the directory pattern ends in a slash, then the default pattern is added to the end (see "Search Parameters" on page 217).

`Search Files Matching Patterns`

Prompts for a search string, root directory and set of filename patterns and then performs a **Root and Patterns** search.

`Search System`

Prompts for a search string and system name and then performs a **System Search**.

`Grep`

Prompts for command line arguments to pass to **`grep`** and then performs a **Grep** search. The **`grep`** command is created from these arguments, with a **`-n`** option and the filename argument **`/dev/null`** as mentioned in "Grep searches" on page 213.

15.2.2.2 Context-dependent searches using menu commands

Edit > Search Files...

Opens a Search Files tool in for a **Plain Directory** or **Root and Patterns** search, using the directory associated with the current tool (in particular, the directory of the buffer displayed in an Editor tool).

If an existing Search Files tool is reused by this command and was last doing a **Root and Patterns** search, then the directory is placed in the **Root Directory** box. Otherwise, the directory is placed in the **Filename Patterns** box for a **Plain Directory** search with the default pattern added to the end (see "Search Parameters" on page 217).

Systems > Search Files...

Prompts for a regexp and performs a **System Search** in the currently selected system.

15.2.2.3 Search History

The Search Files tool keeps a history of previous searches and their results. You can revisit these searches using the and buttons as described in "The history list" on page 30.

15.3 Viewing the results

The results of a search are displayed in the main area of the tool, grouped by file. The file name, the number of matches in that file and the directory are shown. Select a file and expand it to see the line number and text of each line of that file that matches. You can configure the tool to expand the items as they are added as shown in "Display" on page 219.

15.3.1 Displaying in an Editor

Double-click on the filename to open an Editor tool showing that file. Similarly, double-click on the line number to show that line in the Editor. Items that have been edited are shown with a different icon. You can change an item to show as edited or not edited using the **Mark Edited** and **Mark Not Edited** commands on the context menu.

The Editor command **`Next Search Match`** can be used to move to the next item in the last Search Files tool that you used.

15.3.2 Linking to an Editor

You can arrange for an Editor tool to immediately display one of the search matches when you select it. To do this, choose **Link to Editor** from the context menu in the main area of the Search Files tool. To remove the link, choose **Link to Editor** from the context menu again.

Note: this is equivalent to using **Edit > Link from > Search Files 1** in the Editor tool.

15.3.3 Filtering the results

Use the Filter area to restrict the displayed results by a plain string match or a regular expression match, as described in “Filtering information” on page 42.

The filter applies to the text in the match, not to the line number or file names.

15.3.4 Hiding certain results

When there are many results it can be useful to hide some which you know to be uninteresting. Select the lines you wish to hide, raise the context menu and choose **Hide**.

To restore hidden lines to the display, choose **Unhide Others** from the context menu.

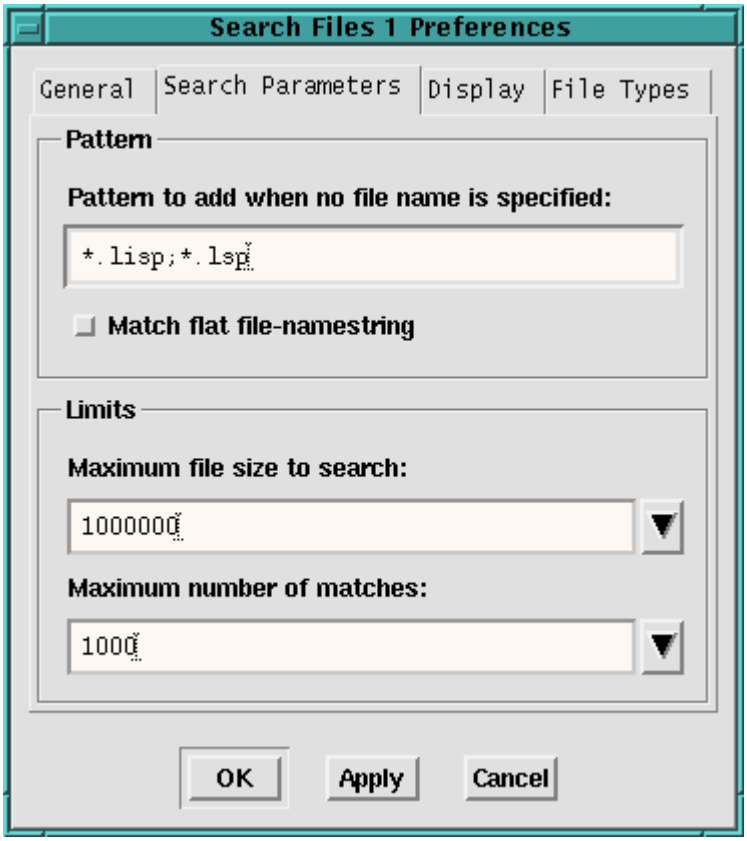
15.4 Configuring the Search Files tool

Various aspects of the Search Files tool's behaviour and display can be configured by selecting **Works > Tools > Preferences...**.

15.4.1 Search Parameters

In the Search Parameters view, you can configure some aspects of searching operations.

Figure 15.4 Setting Search Parameter Preferences



Enter a file name pattern to add when invoking the tool from an Editor command in the **Pattern to add when no file name is specified** box.

Check **Match flat file-namestring** if you want the tool to match filename component of patterns as a flat string, rather than a name and type. If this option is not selected, then any text after the final `.` in the filename is treated as the type and is only matched by similar text after the `.` in the pattern. For example, when **Match flat file-namestring** is not selected, the pattern *dir*`/*p` matches `interp.exe`, where the name `interp` ends

with **p** but does not match **file.lisp**, where the name **file** ends with **e**. Conversely, when **Match flat file-namestring** is selected, *dir*/***p** matches **file.lisp**, where the file-namestring **file.lisp** ends with **p**, but does not match **interp.exe**, where the file-namestring **interp.exe** ends with **e**.

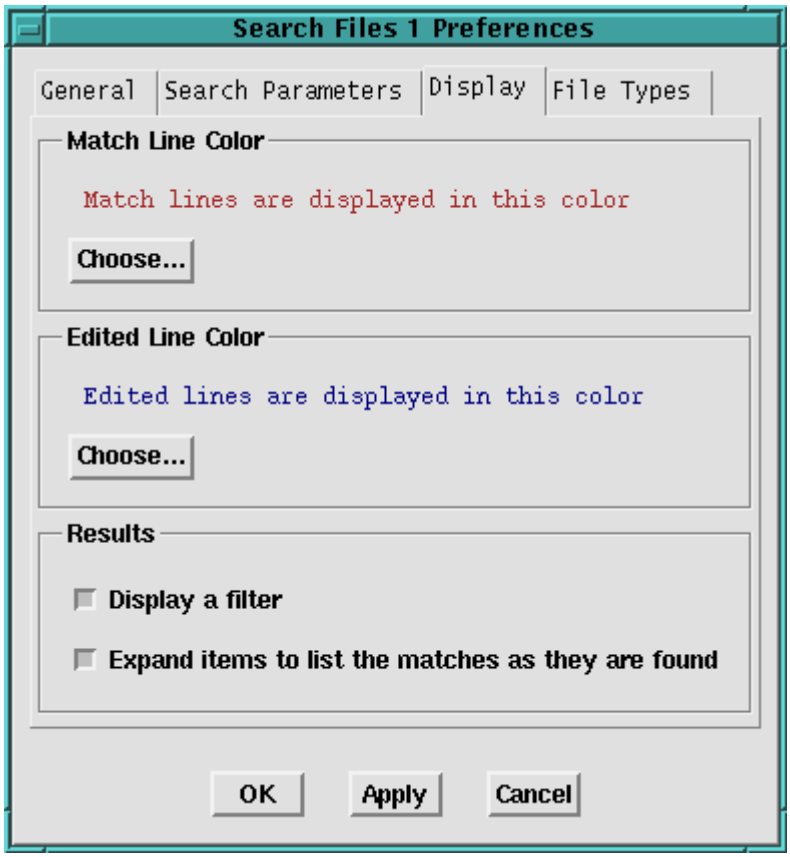
You can specify a limit on the size of files to search in the **Maximum file size to search** box. This limit represents the maximum file size in bytes, and typical values can be selected from the dropdown list.

You can specify a limit on the number of matches displayed by the tool in the **Maximum number of matches** box. Typical values can be selected from the dropdown list. If more matches are found, you are asked whether to stop searching.

15.4.2 Display

You can configure the display of search results using the **Display** view.

Figure 15.5 Setting Display Preferences



Choose a color to display the text of unedited lines that show a match in a file.

Choose a color to display the text of edited lines that show a match in a file.

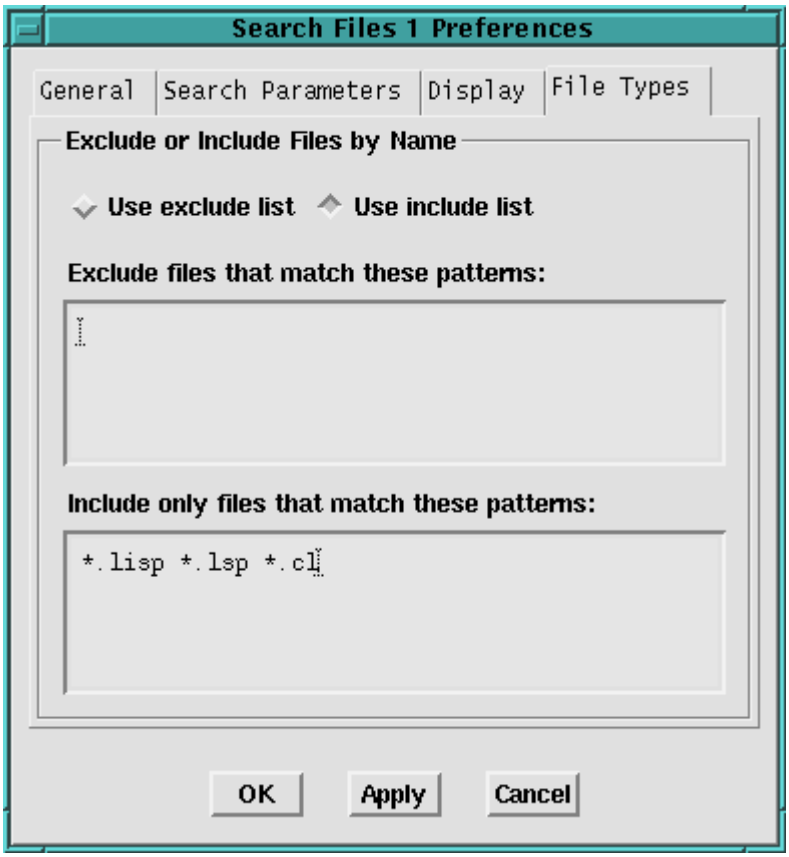
Check **Display a filter** to display a box that can be used to restrict which results are displayed. This shown by default.

Check **Expand items to list the matches as they are found** to cause the items grouped under each file to be expanded while the search is running. The default is to leave them collapsed, allowing you to expand them yourself.

15.4.3 File Types

You can add specify which file types to search in the **File Types** view.

Figure 15.6 Setting File Types Preferences



Check **Use exclude list** if you want to exclude certain file types even though they match the pattern in the **Search Specifications** boxes. Enter the patterns to exclude in the **Exclude** box, with multiple patterns separated by whitespace.

Check **Use include list** if you want to only include certain file types, even if the pattern in the **Search Specifications** should allow other types. Enter the patterns to include in the **Include** box, with multiple patterns separated by whitespace.

You cannot choose both of these options simultaneously.

15.4.4 The External Grep Program

By default, for **Grep** searches the tool runs **grep** on Unix/Linux/FreeBSD/Mac OS X and a specific supplied **grep.exe** on Microsoft Windows. The actual searching utility used can be configured with the variable **lw:*grep-command***.

The arguments passed to the searching utility are constructed using the values of **lw:*grep-fixed-args*** and **lw:*grep-command-format***. It is not be necessary to alter the default values unless you use a non-default value of **lw:*grep-command*** or have a non-standard **grep** installed.

See the *LispWorks Reference Manual* for details of these Search Files tool configuration variables.

16

The Inspector

The Inspector is a tool for examining objects in your Lisp image. You can also use the Inspector to modify the contents of objects, where this is possible.

You can create an Inspector, choose **Works > Tools > Inspector** or click in the Podium.

16.1 Inspecting the current object

It is sometimes more natural to invoke an Inspector on some object you are analysing. You can do this in several ways, including using the **Inspect** menu command.

1. To create an example object, in the Listener, evaluate:

   ```
   (make-instance 'capi:list-panel :items '(1 2 3 4))
   ```

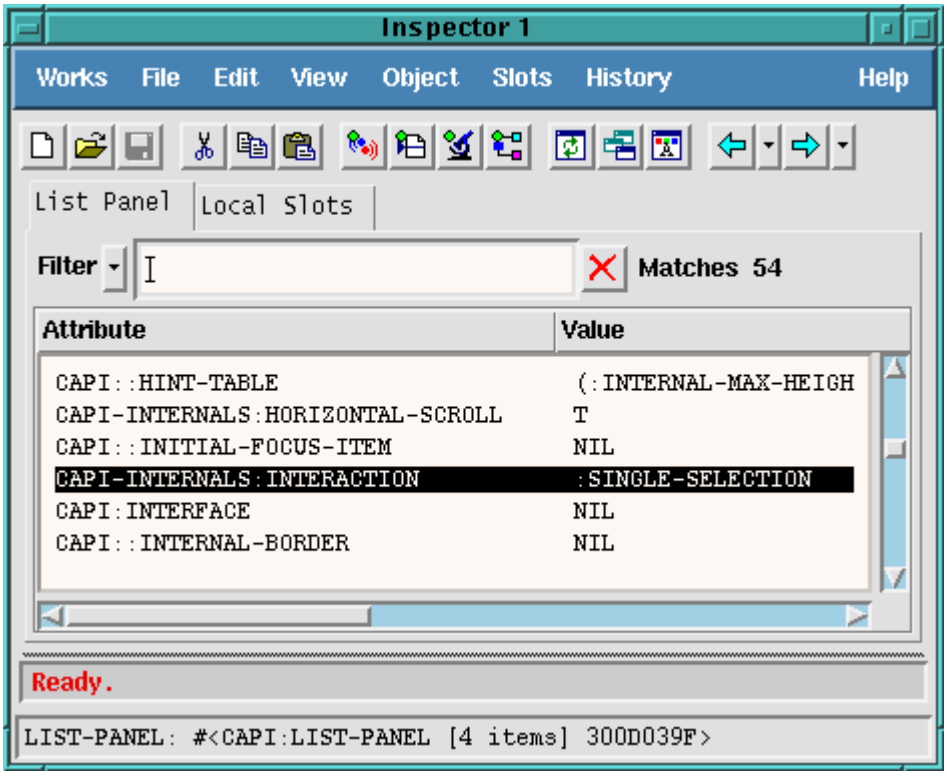
2. Choose **Values > Inspect** from the Listener's menu bar to see the Inspector tool window illustrated in Figure 16.1.

Note that you have not displayed the `list-panel` on screen yet. You will do that in a few minutes.

Another way to inspect the current object (that is, the value of `cl:*`) in the Listener is the keystroke `Ctrl+C Ctrl+I`.

A general way to inspect the current object in most of the LispWorks tools is to click the button.

Figure 16.1 Inspector



16.2 Description of the Inspector tool

The Inspector has the following areas:

- At the top of the Inspector, the tab of the main view shows the name of the object being inspected. There may be other views depending on the type of this object. For class instances, there is a Local Slots view.
- A *Filter area* provides a way of filtering out those parts of an object that you are not interested in.
- A list of attributes and values shows the contents of the object.

16.2.1 Adding a Listener to the Inspector

A small listener pane can be added to the Inspector tool, allowing you to evaluate Common Lisp forms in context, without having to switch back to the main Listener tool itself. To add the listener pane to the Inspector, choose **Show Listener** from the context menu in the attributes and values area.

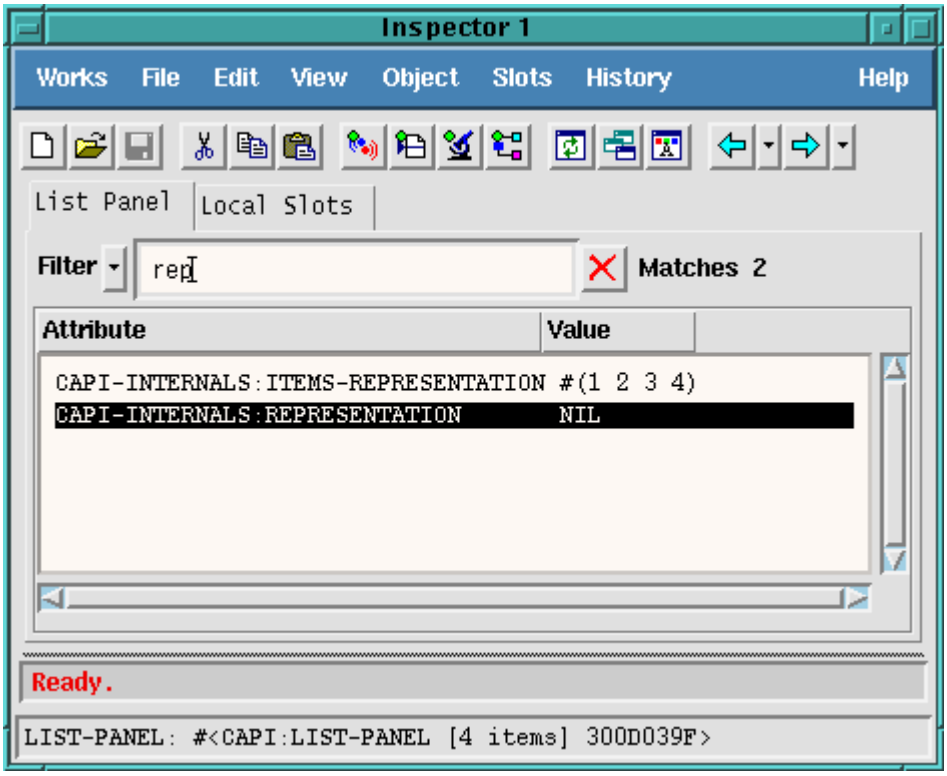
16.3 Filtering the display

Sometimes an object may contain so many items that the list is confusing. If this happens, use the Filter box to limit the display to only those items you are interested in.

This continued example below shows you how to filter the attributes list so that the only slots displayed are those you are interested in.

3. Type **`rep`** in the Filter box.

Figure 16.2 Using filters to limit the display in the Inspector



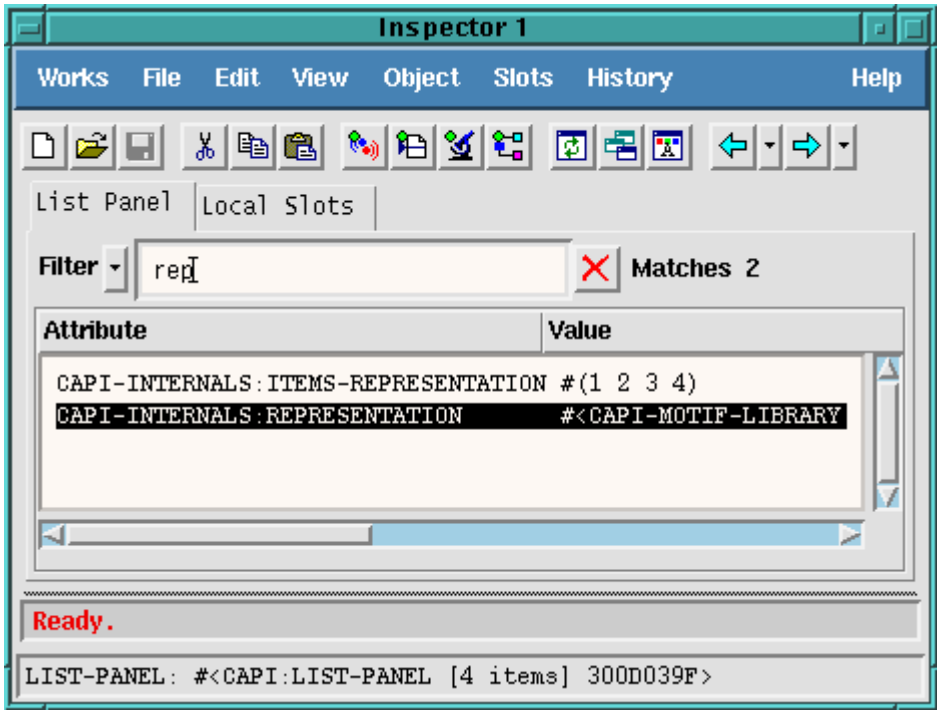
You can immediately see the slots with names that include "rep". The names of the slots, together with their slot values for the object being inspected, are displayed in the attributes list. For example, the representation slot currently contains **`nil`**.

16.3.1 Updating the display

In some circumstances your object might get modified while you are inspecting it, so you should be aware that the inspector display might need to be refreshed. To see this:

4. In the Listener tool call `(capi:contain *)`, where the value of `*` should be the `list-panel` instance that you are inspecting.
5. In the Inspector choose the command **Works > Refresh**. The Inspector should now appear as in Figure 16.3 below.

Figure 16.3 The filtered inspector display, refreshed



Notice that the representation slot no longer has value `nil`. The `list-panel` instance has been modified by calling `capi:contain`, and the Inspector has been updated to show the new slot value.

16.4 Examining objects

The attributes and values list is the most interesting part of the Inspector. Each item in this list describes an attribute of the inspected object by displaying its name (the first field in each entry) and the printed representation of its value (the second field). For

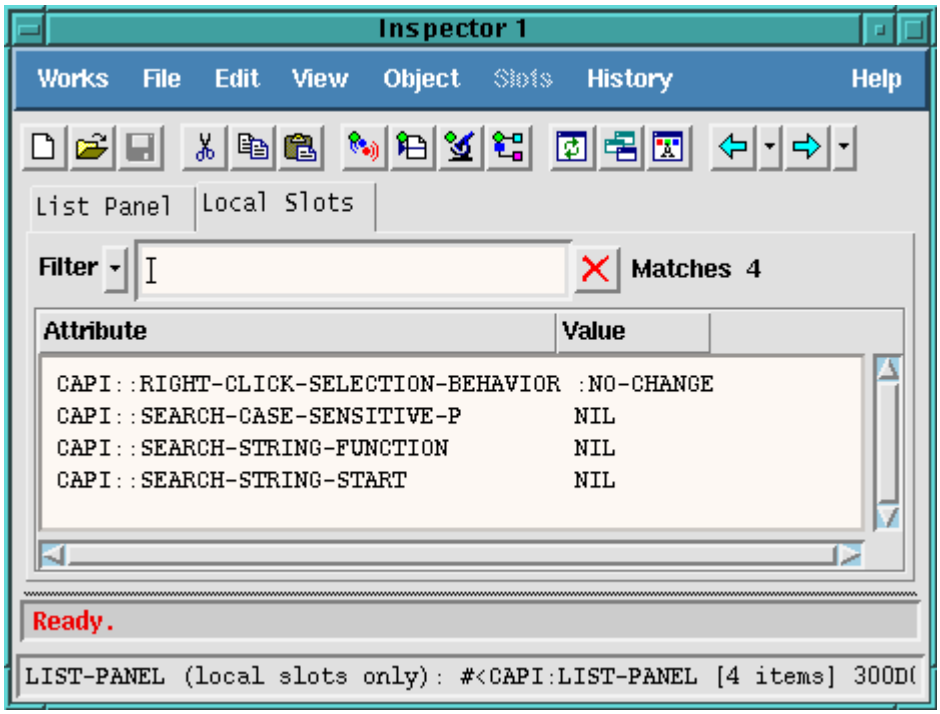
example, the inspection of a CLOS object yields a list of its slots and their values. The description is called an *inspection*.

When inspecting instances of CLOS classes, you can choose to display only those slots which are local to the class. By default, all slots are displayed, including those inherited from superclasses of the class of the inspected object.

6. Press ✕ to remove the filter
7. Select the Local Slots tab

Several slots defined locally for a **`list-panel`** are listed.

Figure 16.4 Inspector showing local slots of a CLOS instance



As well as CLOS instances, other objects including lists and hashtables have multiple views available in the Inspector. For example, a list can be viewed as a plist, alist, cons or list if it has the appropriate structure.

16.5 Operating upon objects and items

The **Object** and **Slots** submenus allow you to perform the standard action commands on either the object being inspected, or the slot values selected in the main list. The commands available are largely identical in both menus, and so are described together in this section.

16.5.1 Examination operations

The standard action commands are available in both the **Object** and **Slots** menus, allowing you to perform a variety of operations on the current object or any items selected in the list. For full details of the standard action commands, see "Performing operations on selected objects" on page 35.

16.5.1.1 Example

Consider the following example, where a closure is defined:

```
(let ((test-button (make-instance 'capi:button)))
  (defun is-button-enabled ()
    (capi:button-enabled test-button)))
```

This has defined the function **is-button-enabled**, which is a closure over the variable **test-button**, where the value of **test-button** is an instance of the **capi:button** class.

1. Enter the definition of the closure shown above into a Listener.
2. Choose **Values > Inspect**.

   The Inspector examines the symbol **is-button-enabled**.
3. Click on the **FUNCTION** slot to select the closure.
4. Choose **Slots > Inspect** to inspect the value in the selected slot.

The closure is inspected.

16.5.1.2 Recursive inspection

You can also double-click on an item in the attributes list to inspect its value. Most users find this the most convenient way to recursively inspect objects.

To return to the previous inspection, choose **History > Previous** or click in the toolbar.

16.5.2 Examining attributes

The **Slots > Attributes** submenu allows you to apply the standard action commands (described in "Operations available" on page 36) to the attributes rather than the values of those attributes.

For example, the **Slots > Attributes > Inspect** command causes the Inspector to view the attributes, rather than the values, of the selected slots. This is useful when inspecting hash tables or lists, since the attributes (keys) might be composite objects themselves.

16.5.3 Tracing slot access

The **Slots > Trace** submenu provides four commands. When inspecting a CLOS object, code which accesses the selected slot may be traced using these commands.

Break on Access causes a break to the debugger if the slot is accessed for read or write, either by a defined accessor or by `slot-value`.

Break on Read causes a break to the debugger if the slot is accessed for read, either by a defined accessor or by `slot-value`.

Break on Write causes a break to the debugger if the slot is accessed for write, either by a defined accessor or by `slot-value`.

Untrace turns off tracing on the selected slot.

The **Object > Trace** submenu provides the same four options, but these commands control the tracing of *all* the slots in the object.

16.5.4 Manipulation operations

As well as examining objects in the Inspector, you can destructively modify the contents of any composite object.

This sort of activity is particularly useful when debugging; you might inspect an object and see that it contains incorrect values. Using the options available you can modify the values in the slots, before continuing execution of a program.

Choose **Slots > Set** to change the value of any selected slots. A dialog appears into which you can type a new value for the items you have selected. Previously entered values are available via a dropdown in this dialog.

Choose **Slots > Paste** to paste the contents of the clipboard into the currently selected items.

16.5.4.1 Example

This example takes you through the process of creating an object, examining its contents, and then modifying the object.

1. Create a button as follows:

   ```
   (setq button1 (make-instance 'capi:button))
   ```

2. Choose **Values > Inspect** in the Listener to inspect the button in the Inspector.

3. In the Listener, use the CAPI accessor **`button-enabled`** to find out whether **`button1`** is enabled.

   ```
   (capi:button-enabled button1)
   ```

   This returns **`t`**. So we see buttons are enabled by default. The next step is to destructively modify **`button1`** so that it is not enabled.

4. Choose **Works > Tools > Preferences...** in the Inspector.

   The Inspector Preferences dialog appears, so that you can change the current package of the Inspector.

5. In the Package box, replace the default package name with **`CAPI`** and click **OK**.

   This changes the process package of the Inspector to the **`CAPI`** package, and the package name disappears from all the slots listed. This makes the display a lot easier to read.

6. In the Inspector, type **`enabled`** into the Filter box.

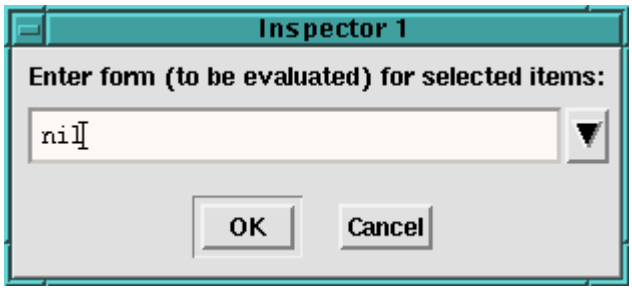
   Button objects have a large number of slots, and so it is easier to filter out the slots that you do not want to see than to search through the whole list. After applying the filter, only one slot is listed.

7. Select the slot **`enabled`**.

8. Choose **Slots > Set...**

A dialog appears into which you can type a new value for the slot **enabled**.

Figure 16.5 Entering a new slot value



9. Note that previously entered forms are available via a dropdown in this dialog. Enter **nil** (or select it from the history) and click on **OK**.

   The attributes and values area shows the new value of the **enabled** slot.

10. Click on the button. This removes the filter and displays all the slots once again.

11. To confirm that the change happened, type the following in the Listener. You should be able to recall the last command using **Alt+P** or **History > Previous**.

    ```
    (capi:button-enabled button1)
    ```

    This now returns **nil**, as expected.

The next part of this example shows you how you can modify the slots of an object by pasting in the contents of the clipboard. This example shows you how to modify the **text** and **font** of **button1**.

12. Type the following into the Listener and then press **Return**:

    ```
    "Hello World!"
    ```

13. Choose **Values > Copy** to copy the string to the clipboard.

14. Select the **TEXT** slot of **button1** in the Inspector.

15. Choose **Slots > Paste** to paste the **"Hello World!"** string into the **text** slot of **button1**.

    This sets the **text** slot of **button1** to the string.

16. Enter the following into the Listener and press **Return**:

```
(let ((font (capi:simple-pane-font button1)))
  (if font
      (gp:find-best-font
       button1
       (apply 'gp:make-font-description
              (append (list :size 30)
                      (gp:font-description-attributes
                       (gp:font-description
                        (capi:simple-pane-font button1))))))
    (gp:make-font-description :size 30)))
```

This form simply calculates a large font object suitable for the button object.

17. Choose **Values > Copy** to copy the font to the clipboard.
18. Select the **FONT** slot of **button1** in the Inspector.
19. Choose **Slots > Paste** to paste the font into the **font** slot of **button1**.
20. Confirm the effect of these changes by displaying the button object. To do this, choose **Object > Listen**.

    This transfers the button object back into the Listener. As feedback, the string representation of the object is printed in the Listener above the current prompt. The object is automatically transferred to the ***** variable so that it can be operated on.
21. In the Listener, type the following:

```
(capi:contain *)
```

This displays a window containing the button object. Note that the text now reads "Hello World!", as you would expect, and that the font size is larger than the default size size for buttons. Note further that you cannot click on the button; it is not enabled. This is because you modified the setting of the **enabled** slot in the earlier part of this example.

16.5.5 Copying in the Inspector

You can easily copy objects in the inspector, ready for pasting into other tools.

To copy the inspected object itself use **Object > Copy**

To copy a slot value use **Slots > Copy**.

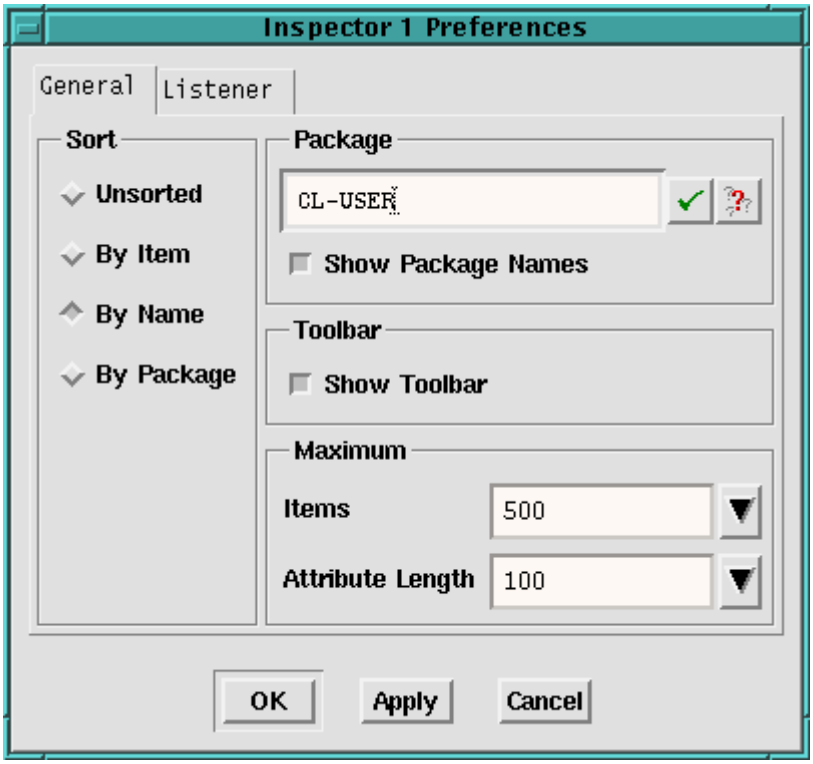
To copy an attribute use **Slots > Attributes > Copy**.

Similarly you can use **Object > ClipSlots > Clip** or **Slots > Attributes > Clip**to place the object itself, a slot value or an attribute on the Object Clipboard, so that you can conveniently retrieve them later. See Chapter 8, “The Object Clipboard” for details.

16.6 Configuring the Inspector

The Inspector Preferences dialog lists a number of different display options in its default general view. These include the standard options for sorting items in the main list, displaying package information and controlling display of the Inspector-specific toolbar, together with some additional options specific to the Inspector.

Figure 16.6 The General tab of the Inspector Preferences dialog



Choose the sort option that you require from those listed in the Sort panel:

By Item — Sorts items alphabetically according to the printed representation of the item.

By Name	Sorts items alphabetically according to their names. This is the default setting.
By Package	Sorts items alphabetically according to the packages of the name field.
Unsorted	Leaves items unsorted. This displays them in the order they were originally defined.

In the Package box, specify the name of the process package for the Inspector. Select **Show Package Names** if you want package names to be displayed in the Inspector. See “Displaying packages” on page 32 for more details.

The Maximum panel contains options to let you configure the amount of information displayed in the Inspector.

Choose a value from the **Attribute Length** drop-down list box to limit the length of any attributes displayed in the main list (that is, the contents of the first column in the list). The default value is 100 characters, and the minimum allowable value is 20 characters.

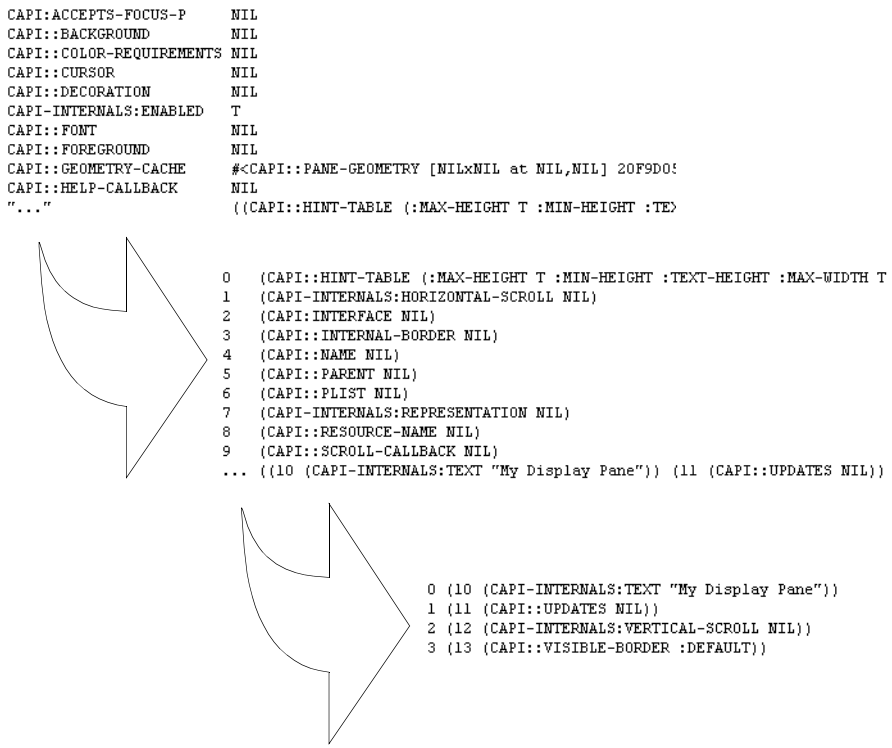
Choose a value from the **Items** drop-down list box to limit the number of items displayed in the main list. By default, 500 items are shown.

If you inspect an object that has more than the maximum number of items, then the excess items are grouped together in a list which itself becomes the last item displayed in the main list. Double-clicking on this inspects the remaining items for the object.

If necessary, the Inspector splits any remaining items into several lists, all linked together in this fashion. For instance, if you limit the maximum number of items to 10, and inspect an object with 24 items, the Inspector displays the first 10, together with an 11th entry, which is a list containing the next ten items. Double-clicking on this

shows the next ten items, together with an 11th entry, which is a list containing the last four items. This is illustrated in Figure 16.7 below.

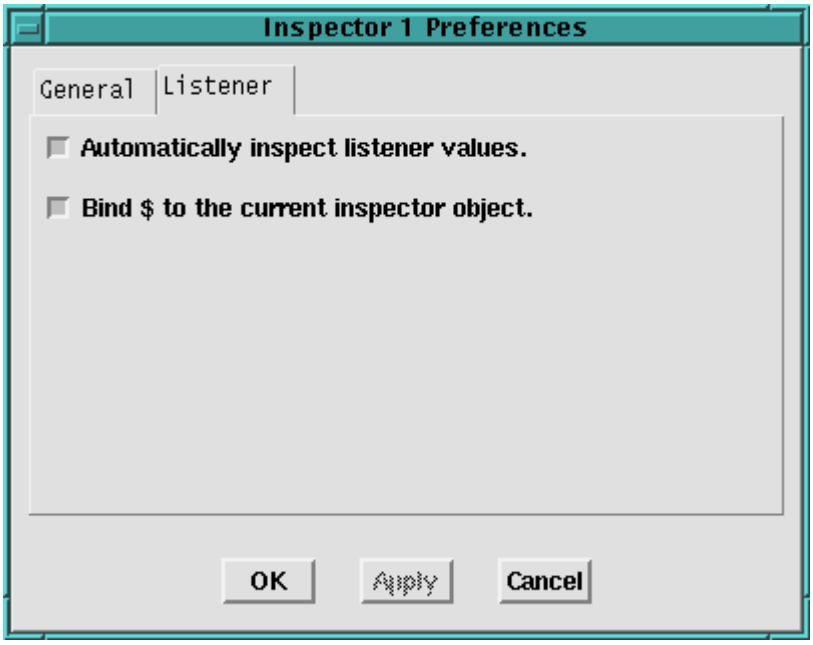
Figure 16.7 Displaying an object with more items than can be displayed



16.7 Customizing the Inspector

The Inspector provides two additional options in the listener view.

Figure 16.8 The Listener tab of the Inspector Preferences dialog



These options control the interaction between the listener pane of the Inspector, if it has one, and the Inspector itself.

See “Adding a Listener to the Inspector” on page 225 for details of how to add a listener pane in the Inspector.

Check **Bind $ to the current inspector object** to bind the variable `$` to the current object in the Inspector in the listener.

Check **Automatically inspect listener values** to inspect listener values automatically.

Both these options are checked by default.

16.8 Creating new inspection formats

There is a default inspection format for each Lisp object.

The Inspector tool can be customized by adding new inspection formats. To do this, you need to define new methods on the generic function **get-inspector-values**. See the *LispWorks User Guide* for a full description.

get-inspector-values takes two arguments: *object* and *mode*, and returns 5 values: *names*, *values*, *getter*, *setter* and *type*.

object — The object to be inspected.

mode — *This* argument should be either **nil** or **eql** to some other symbol. The default format for inspecting any object is its nil format. The nil format is defined for all Lisp objects, but it might not be sufficiently informative for your classes and it may be overridden.

names — The slot-names of *object*.

values — The values of the slots corresponding to *names*. The Inspector displays the *names* and *values* in two columns in the scrollable pane.

getter — This is currently ignored. Use **nil**.

setter — This is a function that takes four arguments: an object (of the same class as *object*), a slot-name, an index (the position of the slot-name in *names*, counting from 0), and finally a new-value. (It is usual to ignore either the slot-name or the index.) This function should be able to change the value of the appropriate slot of the given object to the new-value.

type — This is the message to be displayed in the message area of the Inspector. This is typically either *mode* or—if *mode* is nil—then the name of the class of *object*.

16.8.1 Example

Consider the following implementation of doubly-linked lists.

```
(in-package "DLL")
```

```
(defstruct (dll (:constructor construct-dll)
                (:print-function print-dll))
  previous-cell
  value
  next-cell)

(defun make-dll (&rest list)
  (loop with first-cell
        for element in list
        for previous = nil then cell
        for cell = (construct-dll :previous-cell cell
                                  :value element)
        doing
        (if previous
            (setf (dll-next-cell previous) cell)
          (setq first-cell cell))
        finally
        (return first-cell)))

(defun print-dll (dll stream depth)
  (declare (ignore depth))
  (format stream "#<dll-cell ~A>" (dll-value dll)))
```

You can inspect a single cell by inspecting the following object:

```
(dll::make-dll "mary" "had" "a" "little" "lamb")
```

The resulting Inspector shows three slots: **dll::previous-cell** with value **nil**, **value** with value **"mary"** and **dll::next-cell** with value **#<dll-cell had>**.

In practice, you are more likely to want to inspect the whole doubly-linked list in one window. To do this, define the following method on **get-inspector-values**.

```
(in-package "DLL")

(defun dll-root (object)
  (loop for try = object then next
        for next = (dll-previous-cell try)
        while next
        finally
        (return try)))

(defun dll-cell (object number)
  (loop for count to number
        for cell = object then (dll-next-cell cell)
        finally
        (return cell)))
```

```
(defmethod lw:get-inspector-values ((object dll)
  (mode (eql 'follow-links)))
  (let ((root (dll-root object)))
     (values
       (loop for cell = root then (dll-next-cell cell)
             for count from 0
             while cell
             collecting count)
       (loop for cell = root then (dll-next-cell cell)
      while cell
             collecting (dll-value cell))
       nil
       #'(lambda (object key index new-value)
      (declare (ignore key))
      (setf (dll-value (dll-cell (dll-root object) index)) new-
value))
       "FOLLOW-LINKS")))
```

Inspecting the same object with the new method defined displays a new tab in the Inspector "Follow Links". This shows five slots, numbered from **0** to **4** with values **"mary"** **"had"** **"a"** **"little"** and **"lamb"**.

The following example adds another method to **get-inspector-values** which inspects cells rather than their value slots. The cells are displayed in a "Fllow Cells" tab of Inspector. The setter updates the **next-cell**. Use this new mode to inspect the **"lamb"** cell - that is, double-clink on the **"lamb"** cell in the "Follow Cells" tab - and then set its **next-cell** slot to **(make-dll "with" "mint" "sauce")**.

```
(in-package "DLL")
```

```
(defmethod lw:get-inspector-values
 ((object dll) (mode (eql 'follow-cells)))
  (let ((root (dll-root object)))
     (values
       (loop for cell = root then (dll-next-cell cell)
             for count from 0
             while cell
             collecting count)
       (loop for cell = root then (dll-next-cell cell)
     while cell
             collecting cell)
       nil
       #'(lambda (object key index new-value)
     (declare (ignore key))
     (setf (dll-next-cell (dll-cell (dll-root object) index)) new-
value))
       "FOLLOW-CELLS")))
```

The extended sentence can now be inspected in the **follow-links** mode.

17

The Symbol Browser

17.1 Introduction

The Symbol Browser allows you to view symbols in your LispWorks image found by a match on symbol names, in a manner analogous to the Common Lisp function **apropos** but with additional functionality.

You can restrict the search to specified packages. You can then filter the list of found symbols based on their symbol name, restrict it to those symbols with function or variable definitions and so on, and restrict it based on the symbols' accessibility.

The Symbol Browser also displays information about each selected symbol and allows you to perform operations on the symbol or objects associated with it, including transferring these to other tools in the Common LispWorks IDE by using standard commands.

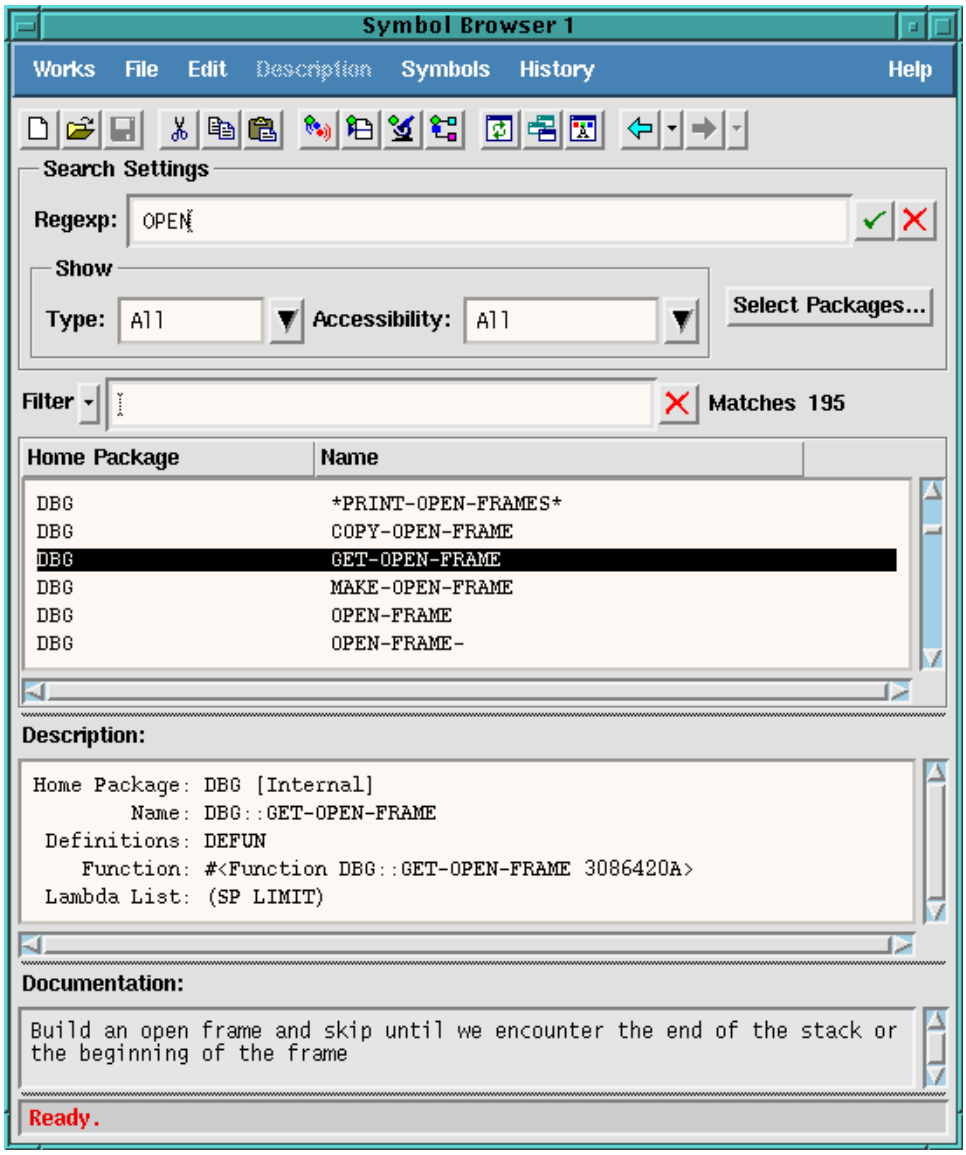
To create a Symbol Browser, choose **Works > Tools > Symbol Browser** or click in the Podium.

Also the editor command **Meta+X Apropos** raises a Symbol Browser tool using the supplied substring to match symbol names.

Also the standard action command **Browse Symbols Like** is available in Context menus and also in the **Expression** menu of editor-based tools. This command raises a Symbol Browser using the current symbol to match symbol names.

17.2 Description of the Symbol Browser

Figure 17.1 The Symbol Browser



The Symbol Browser has five main areas.

17.2.1 Search Settings

The main search setting is the **Regexp:** box.

Enter a string or regular expression in the **Regexp:** box and press **Return** or click the ✓ button. This will match symbol names of interned symbols in a similar way to **apropos**, except that it is a case-insensitive regular expression match.

The remainder of this section describes the other search settings.

17.2.1.1 Packages

By default symbols in all packages are listed, but you can restrict the search to certain packages by clicking the **Select Packages...** button. This raises a dialog which you use in just the same way as the Profiler's Selected Packages dialog - see "Choosing packages" on page 326 for instructions.

When you have selected packages only those symbols whose home package is amongst the selected packages are shown, unless **Accessibility** (see "Accessibility" on page 245) is set to **All**, in which case symbols inherited by the selected packages are also shown.

17.2.1.2 Type

By default all symbols found are displayed but you can restrict this to functions, classes, structures, variables, constants, keywords or others (meaning the complement of all these subsets). If you wish to see, for instance, only those symbols with a function or macro definition then select **Functions** in the **Type** option pane.

17.2.1.3 Accessibility

You can also restrict the display to just those symbols which are present, external or internal in their home package. Select the appropriate item in the **Accessibility** option pane:

All	Show all accessible symbols in the selected packages
Present	Show all present symbols in the selected packages

Externals Only Show only external symbols in the selected packages

Internals Only Show only internal symbols in the selected packages

17.2.2 Filter area

The filter area allows you to filter the display of the symbols list in the same way as other tools. See “Filtering information” on page 42 for details.

17.2.3 Symbols list

The symbols list displays the matched symbol names alongside the name of their home package. You can sort the list by clicking on the **Home Package** or **Name** header at the top of each column.

Select an item in the symbols list to display information about the symbol in the Description and Documentation areas, or to perform an operation on it. You may select multiple symbols, but in this case only the Description and Documentation for the first selected symbol is displayed.

You can transfer the selected symbol or symbols to other tools, for example by **Symbol > Listen** or **Symbol > Inspect**.

To unintern the selected symbol or symbols, choose **Symbol > Unintern...**.

17.2.4 Description area

When you select an item in the Symbols list, various properties of that symbol are displayed in the Description area as appropriate. These can include:

Home Package: The name of the symbol’s home package and an indication of whether it is external or internal

Name: The symbol name

Definitions: The dspec class names for any definitions known to the system

Visible In: The names of the packages (other than the home package) that the symbol is visible in

Function:	The function or macro function
Lambda List:	The lambda list of the function or macro, if known to the system
Plist:	The symbol plist, if non-**`nil`**
Value:	The value of a variable or constant
Class:	The class name, representing the class object

Select an item in the Description list to perform an operation on it. For example, if the symbol has a class definition, you can select the Class: item and do **Description > Listen** to transfer the class object to the Listener tool.

17.2.5 Documentation area

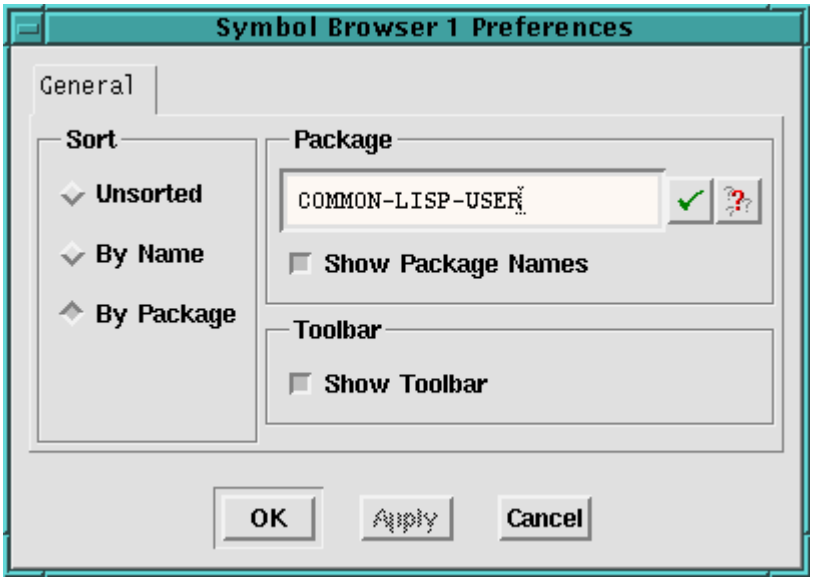
When you select an item in the Symbols list, documentation known to the system is displayed in the Documentation area.

Note: the documentation shown is that returned by the Common Lisp function **`documentation`**.

17.3 Configuring the Symbol Browser

Using the Symbol Browser Preferences dialog, shown in Figure 17.2 below, you can configure some properties of the tool. Choose **Works > Tools > Preferences...** or click to display this dialog.

Figure 17.2 Symbol Browser Preferences dialog



To configure the default sort order for the Symbols list, select **Unsorted**, **By Name** or **By Package** under Sort.

To configure the display of package names in the Description area, alter the Package settings as described in “Displaying packages” on page 32.

You can control whether the Symbol Browser displays the history toolbar by the option **Show Toolbar**, as described in “Toolbar configurations” on page 24.

Press **Apply** to see the effect of your changes in the Symbol Browser tool. Press **OK** to preserve the new settings.

18

The Interface Builder

The Interface Builder helps you to construct graphical user interfaces (GUIs) for your applications. You design and test each window or dialog in your application, and the Interface Builder generates the necessary source code to create the windows you have designed.

You then need to add callbacks to the generated code to connect each window to your application routines.

As you create each window, it is automatically displayed and updated on-screen, so that you can see what you are designing without having to type in, evaluate, or compile large sections of source code.

As well as making code development significantly faster, the Interface Builder allows you to try out different GUI designs, making it easier to ensure that the final design best suits your users' needs.

Note: the Interface Builder is intended for testing interface designs and for generation of the initial versions of the source code that implements your design. It is not suitable for the complete development of complex interfaces. Eventually you should work on the source code directly using the Editor tool (see Chapter 12, "The Editor").

Note: the Interface Builder is available on Windows, Linux and FreeBSD platforms only.

18.1 Description of the Interface Builder

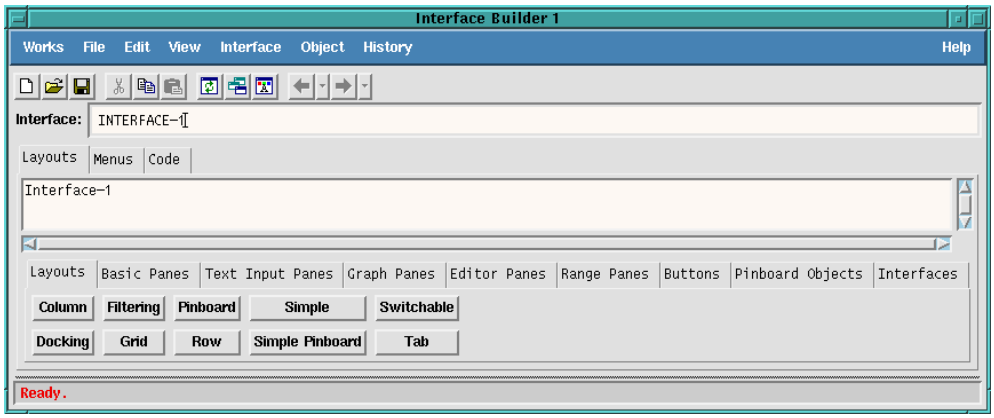
The Interface Builder has three views that help you to design a window.

- The *layouts view* is used to specify the elements in each window or dialog of an application.
- The *menus view* is used to create menus and menu items for each window of an application.
- The *code view* lets you examine the source code that is automatically generated as you create an interface.

The Interface Builder has its own menu bar, containing commands that let you work with a loaded interface, or any of its components.

To create an Interface Builder, choose **Tools > Interface Builder** from the podium.

Figure 18.1 The Interface Builder



Because the Interface Builder generates source code which uses the CAPI library, this chapter assumes at least a minimum knowledge of the CAPI. See the *LispWorks CAPI User Guide* and the *LispWorks CAPI Reference Manual* for details.

A complete example showing you how to use the Interface Builder to design an interface, and how to integrate the design with your own code, is given in Chapter 19, “Example: Using The Interface Builder”. You are strongly advised to work through this example after reading this chapter, or in conjunction with it.

18.2 Creating or loading interfaces

In the context of this chapter, an interface refers to any single window which is used in an application. Thus, an editor, an Open File dialog, or a confirmer containing an error message are all examples of interfaces. The GUI for a complete application is liable to comprise many interfaces. You can load as many different interfaces into the Interface Builder as you like, although you can only work on one interface at once. More formally, the class **`capi:interface`** is the superclass of all CAPI interface classes, which is the set of classes used to create elements for on-screen display. You can load any code which defines instances of this class and its subclasses into the Interface Builder.

Once you have invoked the Interface Builder, you can create new interfaces, or load any that have already been saved in a previous session. You must load or create at least one interface before you can proceed.

18.2.1 Creating a new interface

When you first start the Interface Builder, a new interface is created for you automatically. You can also choose **File > New** or click on to create a new interface. A

blank window, known as the *interface skeleton*, appears on-screen, as shown in Figure 18.2. The interface skeleton contains no layouts or panes, or menus.

Figure 18.2 Skeleton window

You can use **File > New** to create as many interfaces as you want; they are all displayed as soon as you create them. Since you can only work on one interface at a time, use the **History > Items** submenu or the and toolbar buttons to switch between different interfaces that are currently loaded in the Interface Builder.

As an alternative, type the name of an interface directly into the Interface text box and press **`Return`** to create a new interface, or to switch to an interface which is already loaded.

18.2.2 Loading existing interfaces

In the Interface Builder, choose **File > Open...** or click to load an existing interface. You can load any CAPI interface, whether it is one that you have designed using the Interface Builder, or one that has been hand-coded using the CAPI API. You can load as many interfaces as you want, and then use the **History > Items** submenu to swap between the loaded interfaces when working on them.

To load one or more existing interfaces:

1. Ensure the Interface Builder is the active window, and choose **File > Open...**.

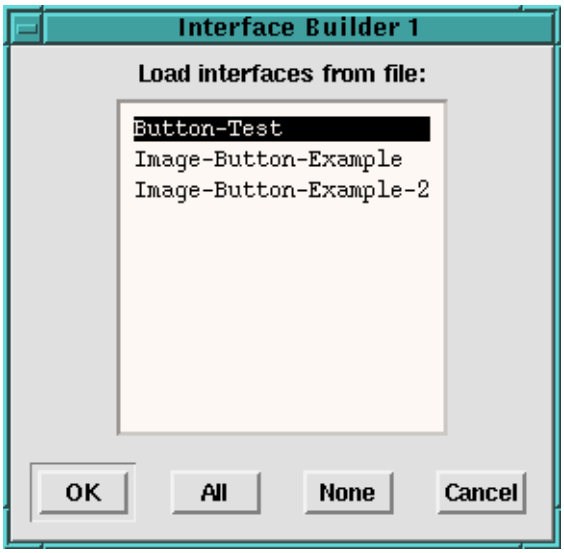
   A file prompter dialog appears.

2. Choose a file of Common Lisp source code.

You should choose a file that contains the source code for at least one CAPI interface. If the file does not contain any such definitions, a dialog appears informing you of this.

Once you have chosen a suitable file, for example the LispWorks library file **`examples/capi/buttons/buttons.lisp`**, a dialog appears listing all the interface definitions that have been found in the file, as shown in Figure 18.3. This lets you choose which interface definitions to load into the Interface Builder. By default, all the definitions are selected. You can select as many or as few of the listed interfaces as you like; the **All** or **None** buttons can help to speed your selection. Click **Cancel** to cancel loading the interfaces altogether.

Figure 18.3 Choosing which interfaces to load into the Interface Builder



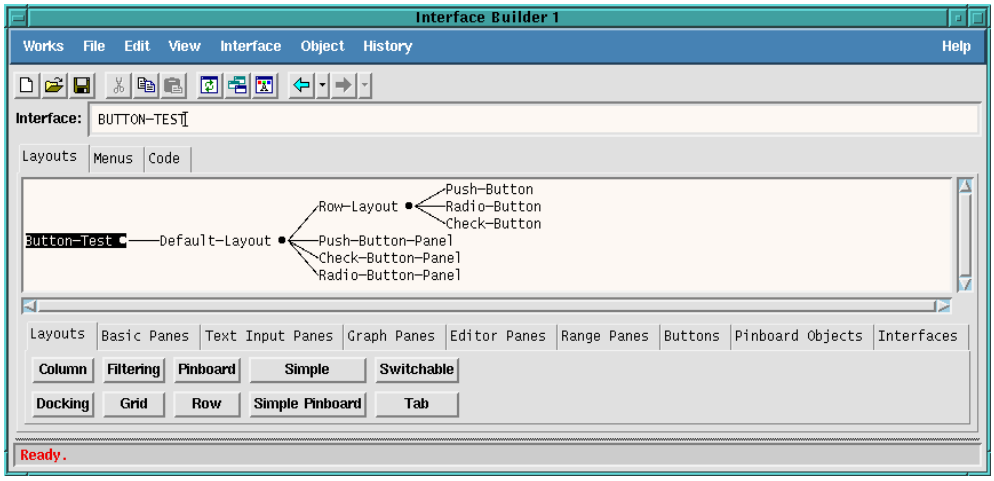
3. Select just the Button-Test interface and click **OK** to load it into the Interface Builder.

Note: the **File > Open...** command in other tools does not display this dialog. To load an interface definition, ensure the Interface Builder window is active.

18.3 Creating an interface layout

The default view in the Interface Builder is the layouts view, as shown in Figure 18.4. You use this view to specify the entire GUI, with the exception of the menus. Click the Layouts tab to swap to this view from any other in the Interface Builder.

Figure 18.4 Displaying the layouts in the Interface Builder



The Interface Builder has three sections in the layouts view.

18.3.1 Interface box

The interface text box displays the name of the current interface; the interface that you are currently working on. Note that there may be several other interfaces loaded into the Interface Builder, but only one can be current.

To switch to another loaded interface, or to create a new interface, type the name of the interface into this area and press **`Return`**. The interface you specify appears and its layouts are shown in the Interface Builder.

While typing in the name of a loaded interface, you can press the **`Tab`** key to complete the interface name if possible. If there is more than one possible completion, a dialog appears so that you can choose the interface to display.

18.3.2 Graph area

This area displays, in graph form, the CAPI elements of the current interface.

By default, the graph is laid out from left to right. The main interface name is shown at the extreme left, and the layouts and elements defined for that interface are shown to the right. The hierarchy of the layouts (that is, which elements are contained in which layouts, and so on) is immediately apparent in the graph.

An item selected in the graph can be operated on by commands in the **Object** menu in the Interface Builder's menu bar. This menu contains the standard action commands described in Section 3.7 on page 35, as well as a number of other commands described throughout this chapter.

To remove a layout or pane from your interface definition, select it in the Graph area and choose **Edit > Cut** or press the toolbar button.

18.3.3 Button panels

At the bottom of the Interface Builder is a tab layouts, each tab of which contains a number of buttons. These tabs list the classes of CAPI elements that can be used in the design of your interface.

- Click the Layouts tab to see the different types of layout that you can use in an interface. This is the default tab and is displayed when you first switch to the layouts view.All other elements must be contained in layouts in order for them to be displayed.
- There are five different types of Panes tab: Basic, Text Input, Graph, Editor and Range. Click on each tab to see the different types of pane that you can use in an interface.
- Click the Buttons tab to see the different types of button that you can use in an interface.
- Click the Pinboard Objects tab to see the different types of pinboard object that you can use in an interface.
- Click the Interfaces tab to see a number of types of pre-defined interface objects that you can use in an interface. These are interfaces which are already used in the Common LispWorks environment, and which may be useful in your own applications.

- Click on the Dividers tab to select between Column and Row layout dividers.

The precise list of items available depends on the package of the current interface. To change this package, choose **Interface > Package...** and specify a package name in the dialog that appears. You must specify a package which already exists in the Lisp image.

Note: The package of the current interface is not necessarily the same as the current package of the Interface Builder. Like all other tools, the Interface Builder has its own current package, which affects the display of symbol names throughout the tool; see "Specifying a package" on page 34 for details. By contrast, the package of the current interface is the package in which the interface is actually defined. The window elements which are available for the current interface depend on the visibility of symbols in that package. By default, both the package of the current interface and the current package of the Interface Builder are set to **`CL-USER`** by default.

An element chosen from any of these areas can be operated on by commands in the **Object** menu. This menu contains the standard action commands described in "Performing operations on selected objects" on page 35.

18.3.4 Adding new elements to the layout

To add a new element to the layout, click the relevant button in any of the tabs in the button panel. The element is added as the child of the currently selected graph node. If nothing is currently selected, the element is added as the child of the *last* selected node.

Because construction of the interface layout is performed by selecting CAPI elements directly, you must be familiar with the way that these elements are used in the construction of an interface.

For instance, the first element to add to an interface is likely to be a CAPI layout element, such as an instance of the **`row-layout`** class or **`column-layout`** class. Not surprisingly, these types of element can be found in the Layouts tab of the button panel. Elements such as buttons or panes (or other layouts) are then added to this layout. In order to generate CAPI interfaces, it is important to understand that all window elements must be arranged inside a layout element in this way.

When you add an element to the design, two windows are updated:

- The graph in the layout view is updated to reflect the position of the new element in the hierarchy.
- The interface skeleton is updated; the element that has been added appears.

When you add an item, an instance of that class is created. By default, the values of certain attributes are set so that the element can be displayed and the hierarchy layout updated in a sensible way. This typically means that name and title attributes are initialized with the name of the element that has been added, together with a numeric suffix. For instance, the first output pane that is added to an interface is called **`Output-Pane-1`**. You should normally change these attribute values to something more sensible, as well as set the values of other attributes. See Section 18.6 for details about this.

For a practical introduction to the process of creating an interface using the Interface Builder, see Chapter 19, “Example: Using The Interface Builder”.

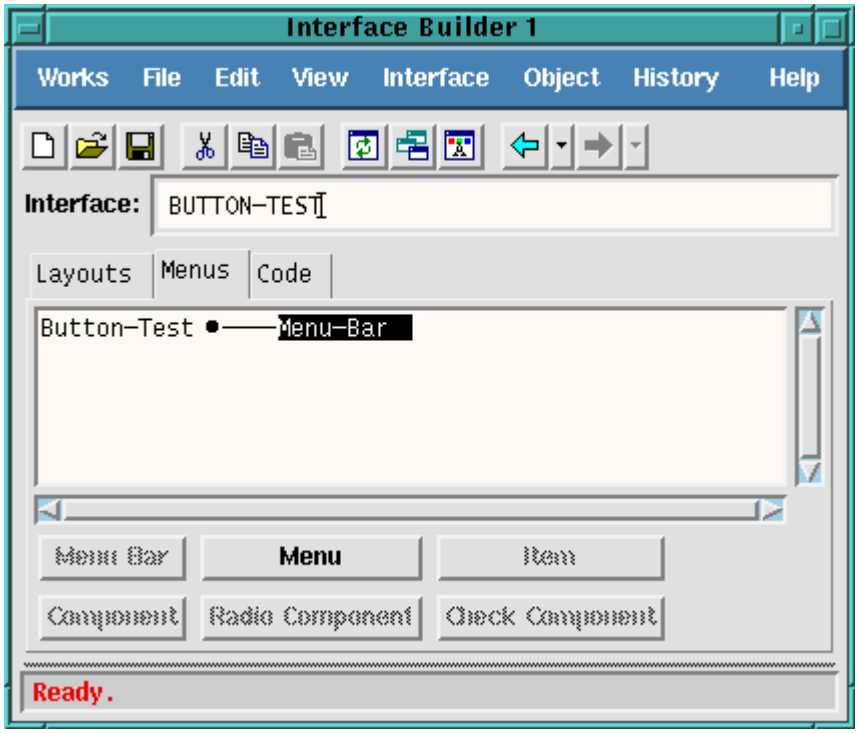
18.3.5 Removing elements from a layout

To remove an element from a layout, select it in the Graph area of the Layouts view and choose **Edit > Cut** or press the toolbar button.

18.4 Creating a menu system

The menus view of the Interface Builder can be used to define a menu system for the current interface. Click the Menus tab to switch to the menus view from any other view in the Interface Builder. The Interface Builder appears as shown in Figure 18.5.

Figure 18.5 Displaying the menu structure of an interface



The menus view has two areas, together with six buttons which are used to create different menu elements. As with layouts, it is important to understand how CAPI menus are constructed. See the *LispWorks CAPI User Guide* for details.

18.4.1 Interface box

This box is identical to the Interface box in the layouts view. See Section 18.3.1 for details.

18.4.2 Graph area

The graph area in the menus view is similar to the graph area in the layouts view. It displays, in graph form, the menu system that has been defined for the current interface. Menu items are displayed as the children of menus or menu components, which in turn are displayed as the children of other menus, or of the entire menu bar.

Like the layouts view, a new menu element is added as the child of the currently selected item in the graph, or the last selected element if nothing is currently selected.

18.4.3 Adding menu bars

A single menu bar is created in any new interface by default. This appears in the graph area as a child of the entire interface.

If you decide to delete the menu bar for any reason, use the **Menu Bar** button to create a new one.

18.4.4 Adding menus

To add a menu, click **Menu** on the button bar at the bottom of the Interface Builder. Each menu must be added as the child of the menu bar, or as the child of another menu or menu component. In the first case, the new menu is visible on the main menu bar of the interface. Otherwise, it appears as a submenu of the relevant menu.

Newly created menus cannot be selected in the interface skeleton until menu items or components are added to them.

By default, new menus are called **`MENU-1`**, **`MENU-2`** and so on, and appear in the interface skeleton as **Menu-1**, **Menu -2** and so on, as relevant. See Section 18.6 for details on how to change these default names.

18.4.5 Adding menu items

To add a menu item to the current interface, click **Item** on the button bar. Each menu item must be added as the child of either a menu or a menu component. If added as the child of a menu component, new items have a type appropriate to that component; see Section 18.4.6 for details.

By default, new menu items are named **ITEM-1**, **ITEM-2**, and so on, and are displayed in the interface skeleton as **Item-1**, **Item-2** and so on, as relevant. See Section 18.6 for details on how to change these default names.

18.4.6 Adding menu components

Menu components are an intermediate layer in the menu hierarchy between menus and menu items, and are used to organize groups of related menu items, so as to provide a better structure in a menu system.

There are three types of menu component which can be defined using CAPI classes:

- Standard menu components.
- Radio components.
- Check components.

18.4.6.1 Standard menu components

A standard menu component can be used to group related menu commands that would otherwise be placed as direct children of the menu bar they populate. This offers several advantages.

- Related menu items (such as **Cut**, **Copy**, and **Paste**) are grouped with respect to their code definitions, as well as their physical location in an interface. This encourages a logical structure which makes for a good design.
- Using standard menu components to group related items is particularly useful when re-arranging a menu system. Groups of items may be moved in one action, rather than moving each item individually.
- Grouping items together using standard menu components adds a separator which improves the physical appearance of any menu.

Click **Component** in the button bar to add a standard menu component to the current interface. Menu components must be added as the children of a menu. Menu components are not visible in the interface skeleton until at least one item or submenu has been added, using the **Item** or **Menu** buttons.

Menu items added to a standard menu component appear as standard menu items in that component.

18.4.6.2 Radio components

A radio component is a special type of menu component, in which one, and only one, menu item is active at any time. For any radio component, **capi:item-selected** always returns **t** for one item , and **nil** for all the others. The menu item that was selected last is the one that returns **t**.

Radio components are used to group together items, only one of which may be chosen at a time.

Click **Radio Component** in the button bar of the Interface Builder to add a radio component to the current interface. Radio components must be added as the children of a menu, and, like standard menu components, are not visible in the interface skeleton until items have been added. To add an item to a radio component, click **Item**. New items are automatically of the correct type for radio components. Note that you cannot add a submenu as an item in a radio component.

The way that a selected radio component is indicated on-screen depends on the operating system or window manager you are running; for example it may be a dot or tick to the left of the selected item. On some systems, a diamond button is placed to the left of every item, and this is depressed for the item which is currently selected.

Like standard menu components, separators divide radio components from other items or components in a given menu.

18.4.6.3 Check components

Like radio components, check components place constraints on the behavior of their child items when selected. For each item in a check component, **capi:item-selected** either returns **t** or **nil**, and repeatedly selecting a given item toggles the value that is returned. Thus, check components allow you to define groups of menu items which can be turned on and off independently.

An example of a check component in the Common LispWorks environment are the commands in the **Tools > Customize** menu, available from any window in the environment.

Click **Check Component** in the button bar of the Interface Builder to add a check component to the current interface. Like other components, check components must be added as the children of a menu, and are not visible until items have been added. Use the **Item** button to add an item to a check component; it is automatically given the

correct menu type. Note that you cannot add a submenu as an item in a radio component.

Like radio components, the way that check components are indicated on-screen depend on the window manager or operating system being used. A tick to the left of any items which are “switched on” is typical. Alternatively, a square button to the left of check component items (depressed for items which are on) may be used.

18.4.7 Removing menu objects

To remove a menu object from your interface definition, select it in the Graph area of the Menus view and choose **Edit > Cut** or press the toolbar button.

18.5 Editing and saving code

As you create an interface in the Interface Builder, source code for the interface is generated. You can use the code view to examine and, if you want, edit this code. You can also save the source code to disk for use in your application. This section discusses how to edit and save the code generated by the Interface Builder, and discusses techniques which let you use the Interface Builder in the most effective way.

18.5.1 Integrating the design with your own code

As your GUI evolves from design into the implmentation phase, you will need to integrate code generated by the Interface Builder with your own code to produce a working application.

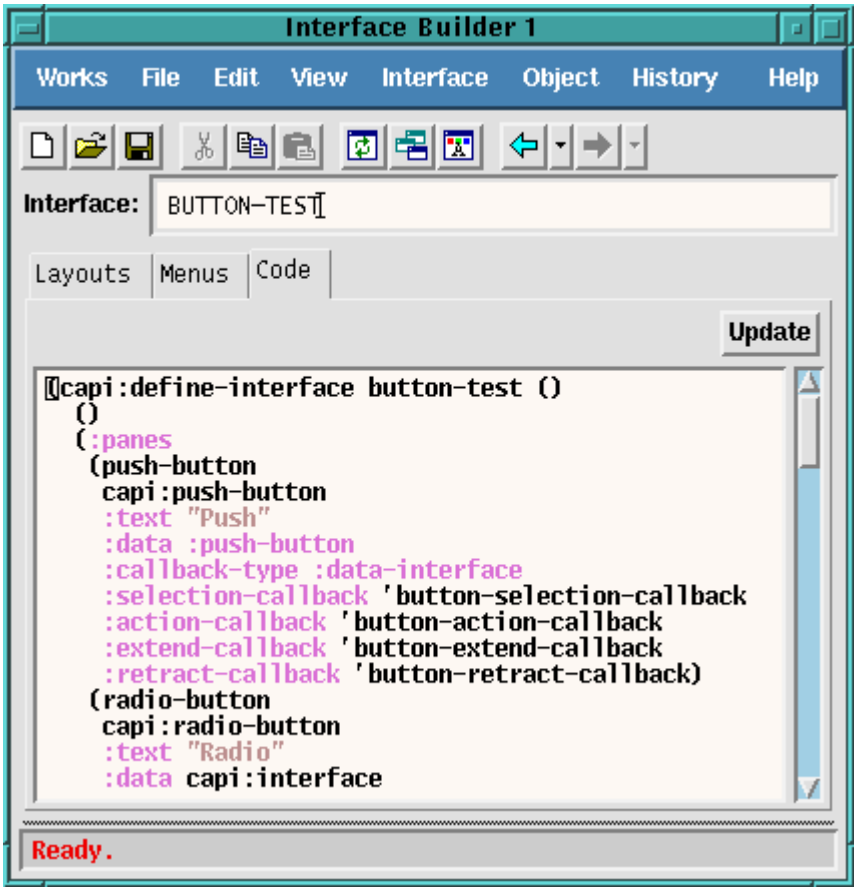
At one extreme, you can attempt to specify the entire GUI for an application using the Interface Builder: even callbacks, keyboard accelerators for menu items, and so on. This way the source code for the entire GUI would be generated automatically. However, this is not the recommended approach.

Instead you should use the Interface Builder for the basic design and initial code generation only. Once you have created an interface skeleton for your window or dialog that you are happy with, augment the automatically-generated source code with hand-written code. At this stage, you will use the Editor tool, rather than the Interface Builder, to develop that window or dialog.

18.5.2 Editing code

Click the Code tab to switch to the code view. You can use this view to display and edit the code that is generated by the Interface Builder. The Interface Builder appears as shown in Figure 18.6.

Figure 18.6 Displaying source code in the Interface Builder



Like the other views in the Interface Builder, an Interface box at the top of the code view displays the name of the current interface. See Section 18.3.1 for details.

The rest of this view is dedicated to an editor window that displays the code generated for the interface. Like other editor windows in Common LispWorks, all the keyboard commands available in the built-in editor are available in the Code area.

If you want to change the definition of the current interface by editing the source code directly, edit the code in the Code area and then click **Update**. Any changes you have made are reflected in the interface skeleton.

18.5.3 Saving code

There are several ways to save the code generated by the Interface Builder into files of source code. Any files that you save are also displayed as buffers in the editor.

Choose **File > Save** or click to save the current interface. If it has already been saved to a file, the new version is saved to the same file. If the interface has not been saved before, you are prompted for a filename. After saving, the file is displayed in the editor.

Choose **File > Save As...** to save the current interface to a specific file. This command always prompts you for a filename; if the interface has not been saved before, this command is identical to **File > Save**, and if the interface has already been saved, this command saves a copy into the file you specify, regardless of the file it was originally saved in. After saving, the file is displayed in the editor.

Choose **File > Save All** to save all of the interfaces that have been modified. A dialog allows you to specify precisely which interfaces to save. Choosing this command is analogous to choosing **File > Save** individually for each of the interfaces you want to save. If there are any interfaces which have not been saved previously, you are prompted for filenames for each one.

Choose **File > Revert to Saved** to revert the current interface to the last version saved.

Choose **File > Close** to close the current interface. You are prompted to save any changes if you have not already done so. The interface name is removed from the **History > Items** submenu.

Individual interface definitions are saved in an intelligent fashion. You can specify the same filename for any number of interfaces without fear of overwriting existing data. Interface definitions which have not already been saved in a given file are added to the end of that file, and existing interface definitions are replaced by their new versions. Source code which does not relate directly to the definition of an interface is ignored.

In this way, you can safely combine the definitions for several interfaces in one file, together with other source code which might be unrelated to the user interface for your application.

Conversely, when loading interfaces into the environment (using **File > Open** or), you do not have to specify filenames which only contain definitions of interfaces. The Interface Builder scans a given file for interface definitions, loads the definitions that you request, and ignores any other code that is in the file. See Section 18.2.2 for details on loading interfaces into the Interface Builder.

This approach to saving and loading interface definitions ensures that your working practices are not restricted in any way when you use the Interface Builder to design a GUI. You have complete control over the management of your source files, and are free to place the source code definitions for different parts of the GUI wherever you want; the Interface Builder can load and save to the files of your choice without failing to load interface definitions and without overwriting parts of the source code which do not relate directly to the GUI.

18.6 Performing operations on objects

There are a large number of operations you can perform on any object selected in the graph of either the layouts view or the menus view. These operations allow you to refine the design of the current interface.

The techniques described in this section apply to an object selected in either the layouts view or the menus view. Any changes made are automatically reflected in both the Interface Builder and the interface skeleton.

18.6.1 Editing the selected object

As in any other Common LispWorks tool, you can use the commands in the **Edit** menu to edit the object currently selected in any graph of the Interface Builder. See Section 3.3 on page 28 for full details on the commands available.

18.6.2 Browsing the selected object

As in other tools, you can transfer any object selected in the graph into a number of different browsers for further examination. The standard action commands that let you do this are available in the **Object** menu. See Section 3.7 on page 35 for details.

18.6.3 Rearranging components in an interface

Rearranging the components of an interface in the most appropriate way is an important part of interface design. This might involve rearranging the layouts and window elements in an interface, or it might involve rearranging the menu system.

The main way to rearrange the components of an interface (either the layouts or the menu components) is to use the cut, copy and paste functions available, as described below.

To move any object (together with its children, if there are any):

1. Select the object in a graph in the Interface Builder (either the layouts view or the menus view, depending on the type of objects you are rearranging).
2. Choose **Edit > Cut** or press .

   The selected object, and any children, are transferred to the clipboard. The objects are removed from the graph in the Interface Builder, and the interface skeleton.
3. Select the object that you want to be the parent of the object you just cut.

   You must make sure you select an appropriate object. For instance, in the Layouts view you must make sure you do not select a window element such as a button panel or output window, since window elements cannot have children. Instead, you should probably select a layout.
4. Choose **Edit > Paste** or press ..

The objects that you transferred to the clipboard are pasted back into the interface design as the children of the newly selected object. The change is immediately visible in both the graph and the interface skeleton.

Note: You can copy whole areas of the design, rather than moving them, by selecting **Edit > Copy** or press instead of **Edit > Paste**. This is useful if you have a number of similar areas in your design.

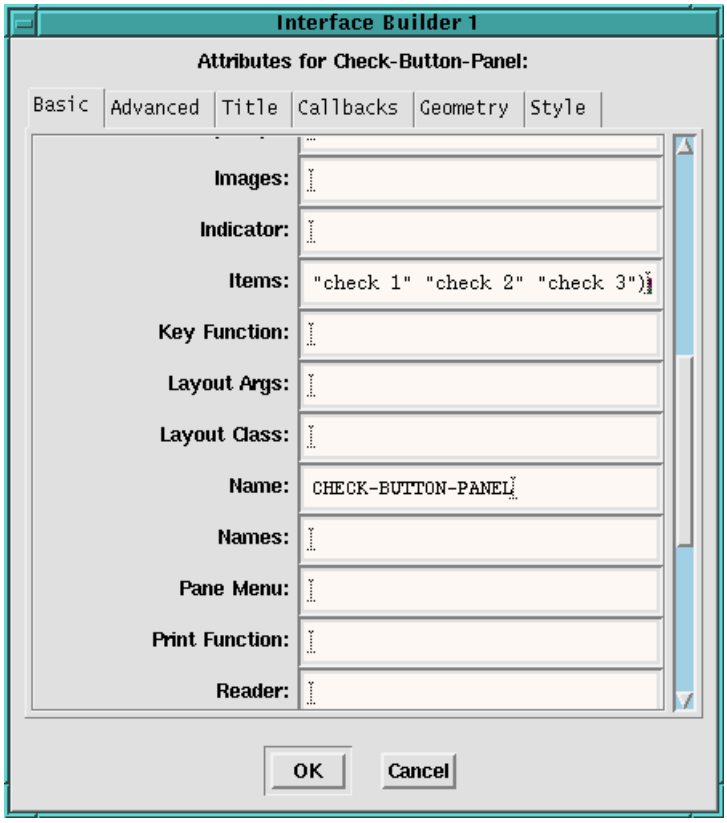
The menu commands **Object > Raise** and **Object > Lower** can be used to raise or lower the position of an element in the interface. This effects the position of the element in the interface skeleton, the layout or menu hierarchy, and the source code definition of the interface. Note that these commands are available from the menu bar in the Interface Builder, rather than from the podium.

18.6.4 Setting the attributes for the selected object

Choose **Object > Attributes** from the Interface Builder's menu bar to display the Attributes dialog for the selected object. This is shown in Figure 18.7. You can also double-click on an object to display this dialog.

The Attributes dialog lets you set any of the attributes available to the selected object, such as symbol names, titles, and callbacks. This gives you a high degree of control over the appearance of any object in the interface.

Figure 18.7 Setting the attributes of the selected object



The precise list of attributes displayed in the dialog depends on the class of the object that you selected in the graph of the Interface Builder.

To set an attribute, type its value into the appropriate text box in the Attributes dialog. Click **OK** to dismiss the Attributes dialog when you have finished setting attribute values.

Because of the large number of attributes which can be set for any class of object, the Attributes dialog shows the attributes in six general categories, as follows:

- Basic attributes.
- Advanced attributes.
- Title attributes.
- Callbacks attributes.
- Geometry attributes.
- Style attributes.

18.6.4.1 Basic attributes

These are the attributes that you are most likely to want to specify new values for. This includes the following information, depending on the class of the selected object:

- The name of the object.
- The items available (for list panels).
- The orientation and borders (for layouts).
- The text representation (for menu items).

18.6.4.2 Advanced attributes

This category lets you specify more advanced attributes of the selected object, such as its property list.

18.6.4.3 Title attributes

This category lets you specify the title attributes of the selected object. These attributes affect the way an object is titled on-screen.

18.6.4.4 Callbacks attributes

This category lets you specify any of the callback types available for the selected object. Many objects do not require any callbacks, and many require several.

18.6.4.5 Geometry attributes

This category lets you control the geometry of the selected object, by specifying any of the available height and width attributes. Geometry attributes are not available for menu objects.

18.6.4.6 Style attributes

This category lets you specify advanced style settings for the selected object. This includes the following attributes:

- The font used to display items in a list.
- The background and foreground colors of an object.
- The mnemonic used for a menu item.

18.7 Performing operations on the current interface

You can perform a number of operations on the current interface, using the commands in the **Interface** menu in the Interface Builder.

18.7.1 Setting attributes for the current interface

Choose **Interface > Attributes** to set any of the attributes for the current interface. An Attributes dialog similar to that shown in Figure 18.7 appears. You set attributes for the current interface in exactly the same way as you do for any selected object in the interface. See Section 18.6.4 for details.

18.7.2 Displaying the current interface

As already mentioned, an interface skeleton is automatically displayed when you load an interface into the Interface Builder, and any changes you make to the design are immediately reflected in the skeleton. There are also a number of commands which

give you more control over the way that the interface appears on-screen as you work on its design.

Choose **Interface > Raise** to bring the interface skeleton to the front of the display. This command is very useful if you have a large number of windows on-screen, and want to locate the interface skeleton quickly.

Choose **Interface > Regenerate** to force a new interface skeleton to be created. The existing interface skeleton is removed from the screen and a new one appears. This command is useful if you have changed the size of the window, and want to see what the default size is; this is especially applicable if you have altered the geometry of any part of the interface while specifying attribute values.

Regenerating the interface is also useful if you set an interface attribute which does not cause the interface skeleton to be updated automatically. This can happen, for instance, if you change the default layout of the interface, which you might want to specify if an interface has several views.

Many interfaces in a GUI are used in the final application as dialogs or confirmers. For such interfaces, the interface skeleton is not necessarily be the most accurate method of display. Choose **Interface > Display as Dialog** or **Interface > Confirmer** to display the current interface as a dialog or as a confirmer, as appropriate. Dialogs are displayed without a menu bar, and with minimal window decoration, so that the window cannot be resized. Confirmers are similar to dialogs, but have **OK** and **Cancel** buttons added to the bottom of the interface. To remove a dialog, click in its Close box.

18.7.3 Arranging objects in a pinboard layout

Most types of layout automatically place their children, so that you do not have to be concerned about the precise arrangement of different objects in an interface. Pinboard layouts, however, allow you to place objects anywhere within the layout.

Objects which are added to a pinboard layout using the Interface Builder have borders drawn around them in the interface skeleton. You can interactively resize and place such objects by selecting and dragging these borders with the mouse.

When you have rearranged the objects in a pinboard layout to your satisfaction, choose **Interface > Display Borders**. This turns off the border display, allowing you to see the appearance of the final interface.

Note: You can only move and resize objects in a pinboard layout when borders are displayed in the interface skeleton. Choosing **Interface > Display Borders** toggles the border display.

18.8 Performing operations on elements

You can transfer any element selected in either the Layouts or Menus views into a number of different browsers for further examination. This is done using the standard actions commands that are available in the **Object** menu. See “Performing operations on selected objects” on page 35 for details. These commands are a useful way of finding out more information about the CAPI objects you use in an interface.

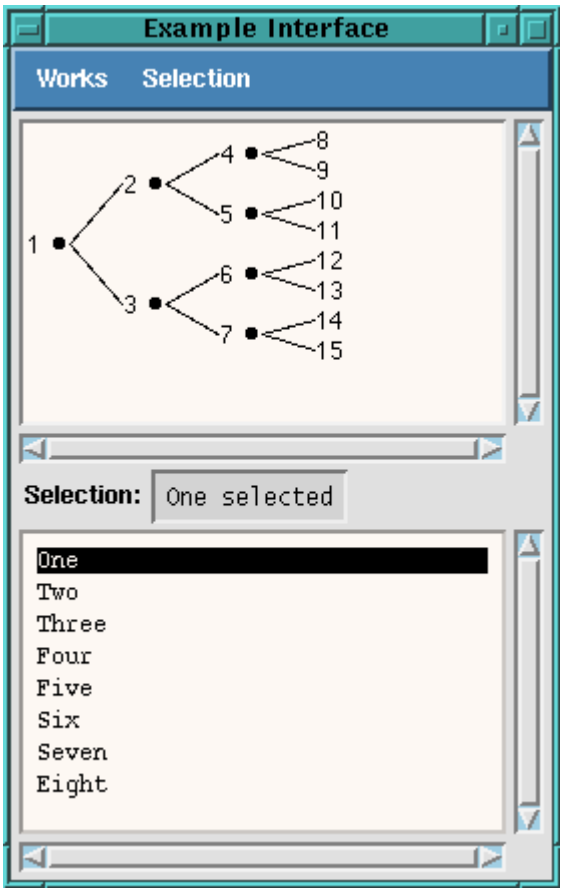
19

Example: Using The Interface Builder

This example shows you how to use the Interface Builder to design a simple interface. It explains how to create the layout and the menu system, and demonstrates some of the attributes that you can set. Finally, the interface is saved to a file, and combined with some other simple code to produce a working example. You are strongly advised to read Chapter 18, “The Interface Builder”, before (or in conjunction with) this chapter. It is also useful, though not essential, if you are familiar with the editor (Chapter 12), the listener (Chapter 20), and Common Lisp systems.

The final interface created is shown in Figure 19.1. It consists of a column layout which contains a graph pane, a display pane, and a list panel.

Figure 19.1 Example interface



Any select action performed in either the graph pane or the list panel is described in the display pane. This includes the following actions:

- Selecting any item
- Deselecting any item
- Extending the selection (by selecting more than one item)

Double-clicking any item in either the graph pane or the list panel displays a dialog which shows which item you double-clicked.

Lastly, there are menu commands available which display, in a dialog, the current selection in either pane. Choose **Selection > Graph** to see the currently selected items in the graph pane, and choose **Selection > List Panel** to see the currently selected items in the list panel.

19.1 Creating the basic layout

This section shows you how to create the basic layout for your interface, without specifying any attributes. Normally, this stage would take you only a few seconds. The process is described in detail here, to illustrate the way that the Interface Builder ensures that the most appropriate item is selected in the graphs of both the layouts and menus views, so as to minimize the steps you need to take when creating an interface.

1. Create an Interface Builder, if you do not already have one.
2. Choose **File > New** or click on the toolbar button.

   A new, empty, interface skeleton appears.
3. If the layouts view is not displayed, click the Layouts tab in the Interface Builder.

   To begin, you need to add the main column layout to the interface using the buttons panels at the bottom of the Interface Builder. The Layouts tab at the bottom of the Interface Builder (as distinct from the Layouts tab you use to switch to the layouts view), lists the different types of layout that you can add to an interface.
4. Click **Column** in the button panel.

   A column layout object is added as a child of the interface object. Nothing appears in the interface skeleton yet, since a column layout is a container for other window objects, and cannot itself be displayed. Note that the column layout remains selected in the layout graph. This is because column layouts are objects which can themselves have children, and the Interface Builder assumes that you are going to add some children next.
5. In the button panel, click the Graph Panes tab, and then click **Graph** to add a graph pane to the interface.

The graph pane object is added as the child of the column layout, and a graph pane appears in the interface skeleton.

6. Next, click the Basic Panes tab and then click **Display**.
7. Next, click **List Panel**.

   The objects that you specify are added to the interface, and the interface skeleton is updated accordingly. Note that the column layout object remains selected throughout. You have now created the basic layout for the interface.

Next, suppose that you decide to add a title to the left of the display pane. You might want to do this make it clear what information is being shown in the display pane.

To do this, you can create a new row layout, add a title pane to it, and then move the existing display pane into this new row-layout. In addition, you must reorganize some of the elements in the interface.

1. Ensure that **`Column-Layout-1`** is still selected in the Layout hierarchy area.

   The new row layout needs to be added as a child of the column layout.
2. In the button panel at the bottom of the Interface Builder, click the Layouts tab to display the available layouts once more.
3. Click on **Row**.

   Notice that the new row layout remains selected, ready for you to add objects to it.
4. Click the Basic Panes tab again, and click **Title**.

   Next, you must move the display pane you have already created, so that it is contained in the new row layout.
5. In the Layout hierarchy area, select **`Display-Pane-1`** and choose **Edit > Cut**.
6. Select **`Row-Layout-1`** and choose **Edit > Paste**.

   The items have already been placed in the row layout in the positions you want them. However, the row layout itself has been added to the bottom of the interface; you want it to be in the same position as the display pane you initially created. To do this, move the list panel to the bottom of the interface.
7. Select **`List-Panel-1`** and choose **Object > Lower** from the menu bar on the Interface Builder itself.

You have now finished creating the layout for the example interface. The next step is to name the elements of the interface in a sensible fashion.

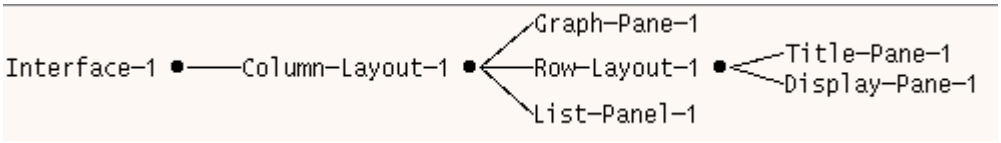
19.2 Specifying attribute values

As you have already seen, the Interface Builder assigns default names such as **`Row-Layout-1`** to the elements you add to an interface; you usually want to replace these with your own names. In addition, there are probably titles that you want to add to the interface; you can see the default titles that have been created by looking at the interface skeleton. The next stage of the example shows you how to change these default names and titles.

Changing the name or title of an element is actually just a case of changing the value assigned to an attribute of that element, as described in Section 18.6.4 on page 267. You would normally assign values to a number of different attributes at once, rather than concentrating on the names and titles of elements. The example is structured in this way to give you an idea of the sort of working practices you might find it useful to adopt when generating interface code.

To recap, the layout hierarchy of the example interface is shown in Figure 19.2. To ensure that you can understand this layout easily in the future, it is important to assign meaningful names and titles to the elements it contains now.

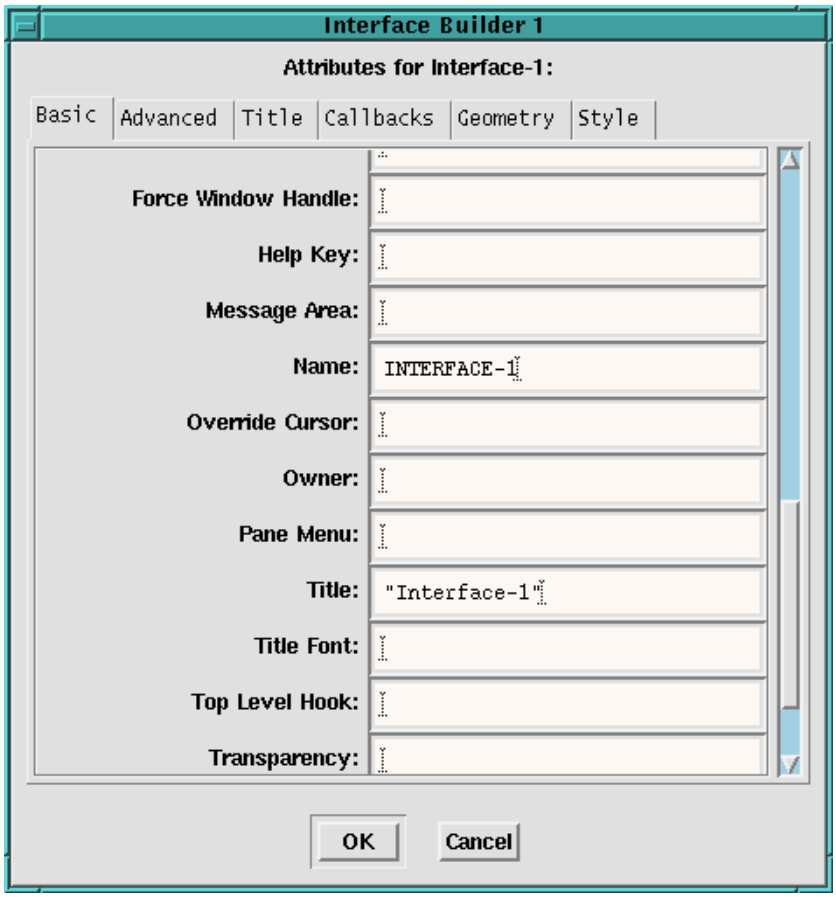
Figure 19.2 Layout hierarchy of the example interface



1. Select the **`Interface-1`** object and then use the **Interface > Attributes** menu item to show the attributes dialog.

The Attributes dialog appears as shown in Figure 19.3.

Figure 19.3 Attributes dialog for the example interface



Notice that the NAME attribute of the interface has the value **`INTERFACE-1`**, and the TITLE attribute has the value **`"Interface-1"`**.

Note: If this is not the first interface you have created in the current session, the number is different.

2. Delete the value in the NAME text box, and type **`ib-example`**.
3. Delete the value in the TITLE text box, and type **`"Example Interface"`**.
4. Click **OK** to dismiss the Attributes dialog and update the interface.

The name of the interface is now displayed as **`lb-Example`** in the Layout hierarchy area, and the title of the interface skeleton changes to **`Example Interface`**.

Note: Case is not significant in the NAME attribute, because it is a Common Lisp symbol, but it is significant in the TITLE attribute, which is a string.

5. Select the **`Column-Layout-1`** element. Double-click to display its Attributes dialog (you will now find this more convenient than using the **Object > Attributes** menu item). Change the value of its NAME attribute to **`main-layout`** and click **OK**.

   Now change the names of the other objects in the interface.

6. Select the graph pane and change its NAME attribute to **`graph`**, and its INTERACTION attribute to **`:extended-selection`**. Click **OK**.

7. Select the list panel and change its NAME attribute to **`list`**, and its INTERACTION attribute to **`:extended-selection`**. Do not click **OK** yet.

   The value of the INTERACTION attribute allows you to select several items from the list panel and the graph pane, using the appropriate method for your platform.

8. Change the ITEMS attribute of the list panel to the following list of strings:

   ```
   '("One" "Two" "Three" "Four" "Five" "Six" "Seven" "Eight")
   ```

9. Click **OK**.

   The row layout you created contains objects which are used solely to display information.

10. Select the row layout object and change its NAME attribute to **`display-layout`**.

11. Change the ADJUST attribute of **`display-layout`** to **`:center`**. Click **OK**.

    This value of the ADJUST attribute centers the title pane and the display pane vertically in the row layout, which ensures their texts line up along the same baseline.

    In the working example, the **`display-layout`** object is going to show information about the current selection, so you need to change the names and titles of the objects it contains accordingly.

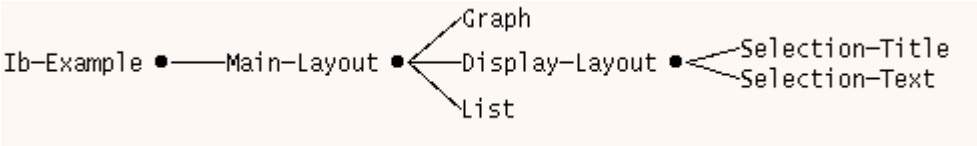
12. Select the title pane and change its NAME attribute to **`selection-title`** and its TEXT attribute to **`"Selection:"`**. Click **OK**.

13. Select the display pane and change its NAME attribute to **`selection-text`**, and its TEXT attribute to **`"Displays current selection"`**. Click **OK**.

    This specifies a text string that is displayed when the interface is initially created. This string disappears as soon as you perform any action in the interface.

The layout hierarchy is now as shown in Figure 19.4. The names that you have assigned to the different objects in the interface make the purpose of each element more obvious.

Figure 19.4 Layout hierarchy with names specified



19.3 Creating the menu system

Next, you need to create a menu system for the example interface. This section shows you how to create the basic objects which comprise it.

1. Click the Menus tab in the Interface Builder to switch to the menus view.

   A menu bar is created automatically when you create a new interface. To create the menu system for the example interface, you need to add a menu which contains two items.

2. Select the **`Menu-Bar`** object in the Menu hierarchy area.

3. Click **Menu** to create the menu, then click **Item** twice to create the two items in the menu.

   Notice that, as in the layouts view, an object remains selected if it can itself have children. This means that creating the basic menu structure is a very quick process.

Next, you need to name the objects you have created. As with the layouts, this is achieved by specifying attribute values.

4. Make sure that the **`Menu-1`** menu is still selected, and use the **Object > Attributes** menu command to display its Attributes dialog.

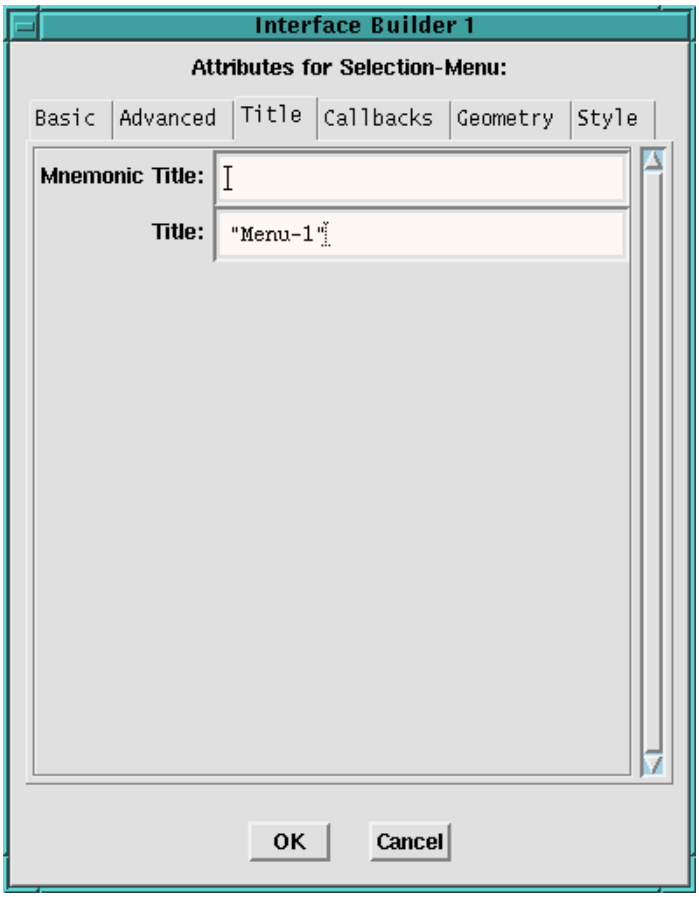
5. Change its NAME attribute to **`selection-menu`**. Do not click **OK** yet.

   As well as specifying the NAME attribute for the menu you created, you need to change the TITLE attribute of each object you created. To do this, you must ensure that the appropriate attribute categories are displayed in the Attributes dialog.

6. Click on the Title tab in the Attributes dialog.

   The Attributes dialog changes to appear as shown in Figure 19.5.

Figure 19.5 Displaying title attributes for a menu



7. In the Title tab view of the Attributes dialog, change the TITLE attribute to **`"Selection"`**. Click **OK**.

   The TITLE attribute is used to specify the title of the menu that appears in the interface itself; note the change in the interface skeleton.

   Next, you need to change the attributes of the two menu items.

8. Select the **`"Item-1"`** object and press **`Return`**.

9. In the Attributes dialog, change the TITLE attribute to **`"Graph"`** and the NAME attribute to **`graph-command`**. Click **OK**.

10. Double-click on the **`"Item-2"`** object to display its Attributes dialog and change the TITLE attribute to **`"List Panel"`** and the NAME attribute to **`list-panel-command`**. Click **OK**.

You have now finished the basic definition of the menu system for your example interface.

19.4 Specifying callbacks in the interface definition

The interface that you have designed contains a complete description of the layouts and menus that are available, but does not yet specify what any of the various elements do. To do this, you need to specify callbacks in the interface definition. As you might expect, this is done by setting attribute values for the appropriate elements in the interface.

In this example, the callbacks that you supply are calls to other functions, the definitions for which are assumed to be available in a separate source code file, and are discussed in Section 19.6. Note that you do not have to take this approach; you can just as easily specify callback functions within the interface definition itself, using lambda notation. It is up to you whether you do this within the Interface Builder, or by loading the code in the editor. If you choose the former, note that it may be easier to use the code view, rather than typing lambda functions into the Attributes dialog.

19.4.1 Specifying layout callbacks and other callback

information

This section shows you how to specify all the callbacks necessary for each element in the example interface, together with other attributes that are required for correct operation of the callback functions. You need to specify attribute values for the display pane, the list panel and the graph pane.

1. If necessary, click the Layouts tab at the top of the Interface Builder to display the layouts view.
2. Select the **`Selection-Text`** display pane and display the Attributes dialog.
3. Set the READER attribute to **`selection-reader`** and click **OK**.

   This reader allows the display pane to be identified by the callback code.

   For the list panel, you need to specify four callbacks and a reader.
4. Select the **`List`** list panel and display its Attributes dialog.
5. Set the READER attribute to **`list-reader`**. Do not click **OK** yet.

   Like the display pane, this reader is necessary so that the list panel can be identified by the callback code.

Next, you need to specify the following four types of callback (make sure you click the "Callbacks" tab):

Selection callback. The function that is called when you select a list item.

Extend callback. The function that is called when you extend the current selection.

Retract callback. The function that is called when you deselect a list item.

Action callback. The function that is called when you double-click on a list item.

6. Now set the following attributes of the list panel. Click **OK** when done.

   SELECTION-CALLBACK to **`'update-selection-select`**
   EXTEND-CALLBACK to **`'update-selection-extend`**
   RETRACT-CALLBACK to **`'update-selection-retract`**
   ACTION-CALLBACK to **`'display-selection-in-dialog`**
7. Select the **`Graph`** graph pane and display its Attributes dialog.

For the graph pane, you need to set the same four callbacks, as well as a reader, and two other attributes that are important for the callback code to run correctly.

8. Set the following attributes of the graph pane.

   SELECTION-CALLBACK to **`'update-selection-select`**
   EXTEND-CALLBACK to **`'update-selection-extend`**
   RETRACT-CALLBACK to **`'update-selection-retract`**
   ACTION-CALLBACK to **`'display-selection-in-dialog`**

9. Set the READER attribute to **`graph-reader`**.

10. Before you set the next callback, evaluate this form:

```
(defun children-function (x)
  (when (< x 8)
    (list (* x 2) (1+ (* x 2)))))
```

    Now set the CHILDREN-FUNCTION attribute to **`'children-function`**.

    The children function defines what is drawn in the graph, and so is vital for any graph pane. It is called when displaying the prototype interface, so it is best to define it before setting this attribute.

11. Click **OK** to dismiss the Attributes dialog..

19.4.2 Specifying menu callbacks

The callbacks that are necessary for the menu system are much simpler than for the layouts; the example interface only contains two menu commands, and they only require one callback each.

1. Click the Menus tab to switch to the menus view.

2. Choose the **`"Graph"`** menu item, display its Attributes dialog and change the CALLBACK attribute to **`'display-graph-selection`**. Click **OK**.

3. Choose the **`"List Panel"`** menu item, display its Attributes dialog and change the CALLBACK attribute to **`'display-list-selection`**. Click **OK**.

19.5 Saving the interface

If you have followed this example from the beginning, the interface is now completely specified. You can now save the source code definition in a file.

1. Choose **File > Save** or click to save the interface definition. Choose a directory in the dialog that appears, and specify the filename **ib-example.lisp** in the "File name" text box.

The file **ib-example.lisp** is displayed in an Editor tool.

19.6 Defining the callbacks

This section shows you how to create the callback functions you need to define in order to complete the working example.

1. Choose **File > New** or click to create a new file.
2. Choose **File > Save** or click to save the file. Save it in the same directory you saved **ib-example.lisp**, and call this new file **ib-callbacks.lisp**.
3. In the editor, specify the package for the callback definitions by typing the following into the **ib-callbacks.lisp** file:

   ```
   (in-package "COMMON-LISP-USER")
   ```

4. Enter the function definitions given in the rest of this section.
5. Choose **File > Save** or click to save the file when you have entered all the function definitions.

The functions that you need to define in this file are divided into the following categories:

- Callbacks to update the display pane.
- Callbacks to display data in a dialog.
- Callbacks for menu items.
- Other miscellaneous functions.

19.6.1 Callbacks to update the display pane

One main function, **update-selection**, serves to update the display pane whenever selections are made in the graph pane or the list panel.

```
(defun update-selection (type data interface)
  (setf (capi:display-pane-text (selection-reader interface))
        (format nil "~A ~A" data type)))
```

The following three functions are the callbacks specified whenever a select, retract or extend action is performed in either the list panel or the graph pane. Each function is named according to the type of callback it is used for, and it simply calls **update-selection** with an additional argument denoting the callback type.

```
(defun update-selection-select (&rest args)
  (apply 'update-selection "selected" args))

(defun update-selection-retract (&rest args)
  (apply 'update-selection "deselected" args))

(defun update-selection-extend (&rest args)
  (apply 'update-selection "extended" args))
```

19.6.2 Callbacks to display data in a dialog

As with **update-selection**, one main function serves to display the data from any action in a dialog.

```
(defun display-in-dialog (type data interface)
  (capi:display-message
   "~S: ~A ~S"
    (capi:interface-title interface) type data))
```

The function **display-selection-in-dialog** is the action callback for both the graph pane and the list panel. It calls **display-in-dialog**, specifying one of the required arguments.

```
(defun display-selection-in-dialog (&rest args)
  (apply 'display-in-dialog "selected" args))
```

Note: Although only one action callback is specified in the example interface, the relevant functions have been defined in this modular way to allow for the possibility of extending the interface. For instance, you may decide at a later date that you want to display the information for an extended selection in a dialog, rather than in the display pane. You could do this by defining a new callback which calls **display-in-dialog**, passing it an appropriate argument.

19.6.3 Callbacks for menu items

Both menu items in the interface need a callback function. As with other callback functions, these are specified by defining a general callback, **display-pane-selection**, which displays, in a dialog, the current selection of any pane.

```
(defun display-pane-selection (reader data interface)
  (declare (ignore data))
  (capi:display-message "~S: ~S selected"
                        (capi:capi-object-name
                         (funcall reader interface))
                        (capi:choice-selected-items
                         (funcall reader interface))))
```

The following two functions call **display-pane-selection**, passing the reader of a pane as an argument. These functions are specified as the callbacks for the two menu items.

```
(defun display-graph-selection (&rest args)
  (apply 'display-pane-selection 'graph-reader args))

(defun display-list-selection (&rest args)
  (apply 'display-pane-selection 'list-reader args))
```

As with the other callback functions, specifying the callbacks in this way allows for easy extension of the example.

19.6.4 Other miscellaneous functions

Graph panes require a function which is used to plot information, called the children function. The value of the ROOTS attribute of a graph is passed as an argument to the children function in order to start the plot. The example interface uses the following simple children function. You already defined this if you have followed the example, but add it also in **ib-callbacks.lisp**:

```
(defun children-function (x)
  (when (< x 8)
    (list (* x 2) (1+ (* x 2)))))
```

Note: The ROOTS attribute of a graph pane has a default value of **(1)**. This is generated automatically by the Interface Builder.

Finally, the function **test-ib-example** is used to create an instance of the example interface.

```
(defun test-ib-example ()
  (capi:display (make-instance 'ib-example
                               :best-height 300
                               :best-width 200)))
```

19.7 Creating a system

If you have followed this example from the beginning, the interface and its callbacks are now completely specified. Next, you can create a Common Lisp system which integrates the interface definition with the callback code.

1. Choose **File > New** or click . This creates a new, unnamed file in the editor.
2. Type the following form into this new file:

   ```
   (defsystem ib-test
     (:package "CL-USER")
     :members ("ib-callbacks" "ib-example"))
   ```

   This form creates a system called **ib-test** that contains two members; **ib-example.lisp** (the file containing the interface definition) and **ib-callbacks.lisp** (the file containing the callback code).
3. Choose **File > Save** or click to save the new file. Save it in the same directory that you saved the **ib-example.lisp** and **ib-callbacks.lisp** files, and call this file **defsys.lisp**.

19.8 Testing the example interface

You have now finished specifying the example interface and its callback functions, so you can test it.

1. Choose **File > Save** or click to save **ib-example.lisp**, **ib-callbacks.lisp**, and **defsys.lisp** if you have not already done so.

   Next, you need to load the **ib-test** system into the environment.
2. In the editor, make sure that the file **defsys.lisp** is visible, and choose **File > Load** to load it and define the system.
3. In the Listener, type the following form.

   ```
   (load-system 'ib-test)
   ```

   The **ib-test** system, together with its members, is loaded.
4. To test the interface, type the following form into the listener.

   ```
   (cl-user::test-ib-example)
   ```

A fully functional instance of the example interface is created for you to experiment with.

20

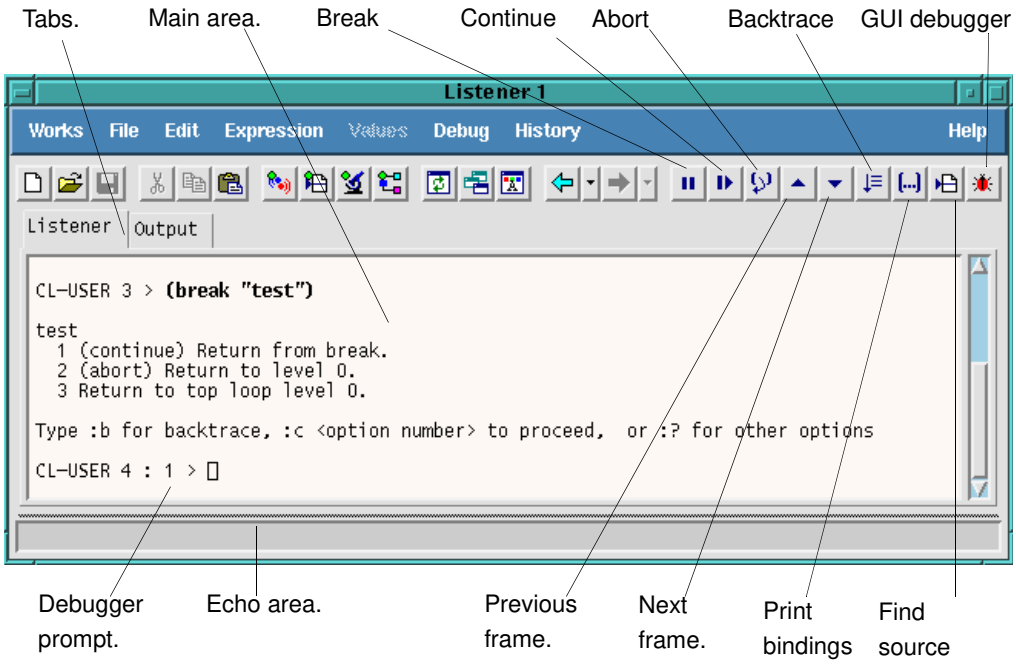
The Listener

The Listener is a tool that lets you evaluate Common Lisp expressions interactively and immediately see the results. It is useful for executing short pieces of Common Lisp, and extensive use is made of it in the examples given in this manual. This chapter describes all the facilities of the Listener.

20.1 The basic features of a Listener

A Listener is created automatically when you start Common LispWorks. You can also create a Listener yourself by choosing **Works > Tools > Listener** or press in the Podium.

Figure 20.1 Listener



In the Listener view, the main area of the Listener contains a prompt at the left side of the window.

Rather like the other types of command line prompt, this prompt helps you identify the point in the Listener at which anything you type is evaluated. It may also contain other useful information, by default this is the current package and the current number in the command history list. If your Listener is in the debugger, as Figure 20.1, the prompt also contains a colon followed by an integer indicating how many debugger levels have been entered.

Throughout this chapter, the prompt is shown in pieces of example code as *PROMPT* >.

You can click the Output tab to display the output view of the Listener; this view displays any output that is created by the Listener, or any child processes created from the Listener..

To familiarize yourself with the Listener, follow the instructions in the rest of this chapter, which forms a short lesson. Note that, depending on the nature of the image you are using, and the configuration that the image has been saved with, the messages displayed by Lisp may be different to those shown here.

20.2 Evaluating simple forms

1. Type the number **12** at the prompt, and press **Return**.

In general, assume that you should press **Return** after typing something at the prompt, and that you should type at the *current prompt* (that is, the one at the bottom of the screen). In fact, the latter is not always necessary; "Execute mode" on page 298 describes how to move the cursor to different places, and thus you may not always be on the current prompt.

Any Common Lisp form entered at the prompt is evaluated and its results are printed immediately below in the Listener.

When Common Lisp evaluates a number, the result is the number itself, and so **12** is printed out:

```
PROMPT > 12
12

PROMPT >
```

When results are printed in the Listener, they start on the line following the last line of input. The **12** has been printed immediately below the first prompt, and below that, another prompt has been printed.

2. Type ***** at the current prompt.

```
PROMPT > *
12

PROMPT >
```

The variable ***** always has as its value the result of the previous expression; in this case, **12**, which was the result of the expression typed at the first prompt. For a full description, see the Common Lisp Hyperspec. This is an HTML version of the ANSI Common Lisp standard which is supplied with LispWorks.

3. Type **(setq val 12)** at the current prompt.

   ```
   PROMPT > (setq val 12)
   12

   PROMPT >
   ```

   The expression sets the variable **val** to **12**. The result of evaluating the form is the value to which **val** has been set, and thus the Listener prints **12** below the form typed at the prompt.

   This is exactly the same behavior as before, when you typed a number it was evaluated and the result printed in the Listener. What is different this time, of course, is that Lisp has been told to "remember" that **12** is associated with **val**.

4. Type **val**.

   The form is evaluated and **12** is printed below it.

5. Type **(+ val val val)**.

   The form, which computes the sum of three **val**s, is evaluated, and **36** is printed below it.

20.3 Re-evaluating forms

If you change **val** to some other number, and want to know the sum of three **val**s again, you can avoid re-typing the form which computes it. To see how this is done, follow the instructions below.

1. Type **(setq val 1)**.

   The variable **val** is now set to 1.

2. Press **Meta+P** or choose **History > Previous** or click .

   ```
   PROMPT > (setq val 1)
   ```

   The form you previously typed appears at the prompt. At this point, you could edit this form and press **Return** to evaluate the edited form. For the moment, just carry on with the next instruction.

3. Press **Meta+P** again, and then press **Return**.

```
PROMPT > (+ val val val)
3

PROMPT >
```

Pressing **Meta+P** a second time displayed the second to last form that you evaluated. This time, pressing **Return** immediately afterwards simply re-evaluates the form. Note that you could have edited the recalled form before evaluating it. You can use **Meta+P** repeatedly, recalling any form that you have evaluated in the current session.

This time the form evaluates to the number **3**, because the value of **val** was changed in the interim.

20.4 Interrupting evaluation

The button ❚❚ interrupts evaluation in the Listener. The key stroke **Alt+Ctrl+C** and the **Interrupt Lisp** button in the GC Monitor window can also be used.

This is useful for stopping execution in the middle of a loop, or for debugging. When the interrupt is processed, the debugger is entered, with a continue restart available.

20.5 The History menu

The forms and commands typed at previous prompts are stored in the *history list* of the Listener. It is so named because it records all the forms and commands you have typed into the Listener. Many other command line systems have a similar concept of a history. Each form or command in the history is known as an *event*.

You can obtain a list of up to the last ten events in the history by displaying the **History > Items** menu. To bring a previous event to the prompt, choose it from this menu.

For more information about history lists in the environment, see “The history list” on page 30.

20.6 The Expression menu

The **Expression** menu lets you perform operations on the current expression, that is, the symbol in which the cursor currently lies. It behaves in exactly the same way as

the **Expression** menu in the Editor tool. See "Current buffers, definitions and expression" on page 179 for details.

Choose **Expression > Class** to look at the class of the current expression in a Class Browser. See Chapter 7, "The Class Browser" for full details about this tool.

Choose **Expression > Find Source** to search for the source code definition of the current expression. If the definition is found, the file is displayed in the editor and the definition is highlighted. See Chapter 12, "The Editor" for an introduction to the editor. Note that you can find only the definitions of symbols you have defined yourself — those for which you have evaluated or compiled the source code — not those provided by the system.

Choose **Expression > Documentation** to display the Common Lisp documentation (that is, the result of the function **`documentation`**) for the current expression. If such documentation exists, it is printed in a help window.

Choose **Expression > Arguments** to print the lambda list of the current expression in the echo area, if it is a function, generic function or method. This is similar to using the keystroke **`Meta+=`**, except that the current expression is automatically used.

Choose **Expression > Value** to display the value of the current expression in the echo area.

Choose **Expression > Inspect Value** to inspect the value of the current expression in the Inspector tool. If the value is **`nil`**, a message is printed in the echo area.

Choose **Expression > Toggle Breakpoint** to add or remove a stepper breakpoint on the current expression. See for information about using the Stepper tool.

Choose **Expression > Evaluate Region** to evaluate the Lisp code in the current region. You must make sure you have marked a region before choosing this command; see "Marking the region" on page 171. Whether you use the mouse or keyboard commands to mark a region does not matter.

Choose **Expression > Compile Region** to compile the Lisp code in the current region.

Choose **Expression > Macroexpand** to macroexpand the current form. The macroexpansion is printed in the output view, which is displayed automatically. Click the Output tab to redisplay the output at any time.

Choose **Expression > Walk** to walk the current form. This performs a recursive macroexpansion on the form. The macroexpansion is printed in the output view, which is displayed automatically. Click the Output tab to redisplay the output at any time.

Choose **Expression > Trace** to display a menu of trace commands which can be applied to the current expression. See "Tracing symbols from tools" on page 40 for full details.

Choose **Expression > Function Calls** to browse the current expression in a Function Call Browser. See Chapter 13, "The Function Call Browser" for more details.

Choose **Expression > Generic Function** to browse the current expression in a Generic Function Browser. This command is only available if the current expression is a generic function. See Chapter 14, "The Generic Function Browser" for more details.

Choose **Expression > Browse Symbols Like** to view symbols containing the current expression in a Symbol Browser. This command is analogous to **`cl:apropos`**. See "The Symbol Browser" on page 243for more details.

20.7 The Values menu

The **Values** menu lets you perform operations on the results of the last expression entered at the Listener prompt. The values returned from this expression are referred to as the *current values*.

The menu is not available if the most recent input was not a Common Lisp form. This is because the evaluation of the last expression entered must have produced at least one value to work on.

The **Values** menu gives you access to the standard action commands described in "Performing operations on selected objects" on page 35.

Note that the most commonly used of the standard action commands are available from the toolbar. For instance, to inspect the current values, click the button.

20.8 The Debug menu

This menu allows you to perform command line debugger operations upon the current stack frame. The menu is only available when the debugger has been invoked by some activity within the Listener.

Some of the most commonly-used command line debugger commands are available from the **Debug** menu. You can also invoke the debugger tool from this menu.

Choose **Debug > Restarts** to display a submenu containing all the possible restarts for the debugger, including the abort and continue restarts. Choose any of the commands on this submenu to invoke the appropriate restart. Note that the continue and abort restarts are also available on the toolbar.

Choose **Debug > Listener > Backtrace** to produce a backtrace of the error.

Choose **Debug > Listener > Bindings** to display information about the current stack frame.

Choose **Debug > Frame > Find Source** to find the source code definition of the function at the current call frame and display it in an editor.

Choose **Debug > Listener > Next** to move to the next call frame in the stack.

Choose **Debug > Listener > Previous** to move to the previous call frame in the stack.

Choose **Debug > Start GUI Debugger** to invoke a debugger tool on the current error. See Chapter 10, “The Debugger Tool”, for full details about using this tool.

Choose **Debug > Report Bug** to report a bug in LispWorks.

You can also invoke any of the commands from this menu by typing keyboard commands into the Listener itself. See the *LispWorks User Guide* for more details.

20.9 Execute mode

The Listener is actually a special type of editor window, which is run in a mode known as *execute mode*. This means that, as well as the normal keyboard commands available to the editor, a number of additional commands are available which are especially useful when working interactively.

20.9.1 History commands

These commands are useful in the common situation where you need to repeat a previously entered command, or enter a variant of it.

History First *Editor Command*

Emacs Key Sequence: **Ctrl+C <**

Replaces the current command by the first command.

History Kill Current *Editor Command*

Emacs Key Sequence: **Ctrl+C Ctrl+K**

Kills the current command when in a listener.

History Last *Editor Command*

Emacs Key Sequence: **Ctrl+C >**

Replaces the current command by the last command.

History Next *Editor Command*

Emacs Key Sequence: **Meta+N** or **Ctrl+C Ctrl+N**

Displays the next event on the history list. That is, it replaces the current command by the next one. This is not available if you are at the end of the history list. In KDE/Gnome editor emulation, this **History Next** command is bound to **Ctrl+Down**.

History Previous *Editor Command*

Emacs Key Sequence: **Meta+P** or **Ctrl+C Ctrl+P**

Displays the previous event on the history list: that is, it replaces the current command by the previous one. In KDE/Gnome editor emulation, this **History Previous** command is bound to **Ctrl+Up**.

History Search *Editor Command*

Emacs Key Sequence: **Meta+R** or **Ctrl+C Ctrl+R**

Searches for a previous command containing a given string, which it prompts for, and replaces the current command with it.

History Search From Input *Editor Command*

Emacs Key Sequence: None

Searches the history list using current input. That is, it searches for a previous command containing the string entered so far, and replaces the current command with it.

Repeated uses step back to previous matches.

History Select *Editor Command*

Emacs Key Sequence: **`Ctrl+C Ctrl+F`**

Presents a list of items in the command history, and replaces the current command with the selection.

History Yank *Editor Command*

Emacs Key Sequence: **`Ctrl+C Ctrl+Y`**

Inserts the previous command into the current one, when in a listener.

20.9.2 Debugger commands

These commands are useful when in the debugger in the Listener:

Debugger Backtrace *Editor Command*

Emacs Key Sequence: **`Meta+Shift+B`**

Gets a backtrace when in the debugger.

Debugger Abort *Editor Command*

Emacs Key Sequence: **`Meta+Shift+A`**

Aborts in the debugger.

Debugger Continue *Editor Command*

Emacs Key Sequence: **Meta+Shift+C**

Continues in the debugger.

Debugger Previous *Editor Command*

Emacs Key Sequence: **Meta+Shift+P**

Displays the previous frame in the debugger.

Debugger Next *Editor Command*

Emacs Key Sequence: **Meta+Shift+N**

Displays the next frame in the debugger.

Debugger Edit *Editor Command*

Emacs Key Sequence: **Meta+Shift+E**

Edits the current frame in the debugger.

Debugger Print *Editor Command*

Emacs Key Sequence: **Meta+Shift+V**

Prints the variables of the current frame in the debugger.

20.9.3 Miscellaneous Listener commands

Here are more commands, with their Execute mode key bindings, which are useful in the Listener

Inspect Star *Editor Command*

Emacs Key Sequence: **Ctrl+C Ctrl+I**

Inspects the current value (that is, the value of the Common Lisp variable *****).

Inspect Variable *Editor Command*

Emacs Key Sequence: None

Inspects the value of an editor variable, which is prompted for.

Throw To Top Level *Editor Command*

Emacs Key Sequence: **Meta+K**

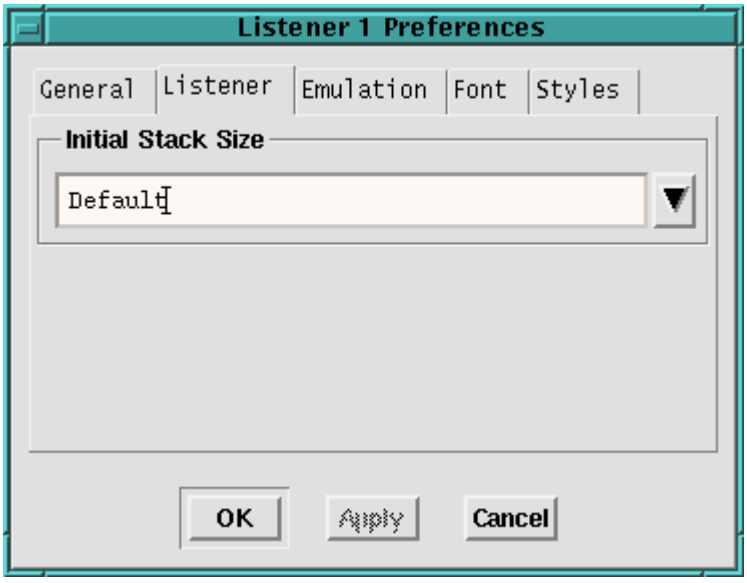
Abandons the current input.

For more details about other keyboard commands available in the editor, see Chapter 12, "The Editor", and the *LispWorks Editor User Guide*.

20.10 Setting Listener preferences

To set options for the Listener tool choose **Works > Tools > Preferences...** or click within an Listener to display the Listener Preferences dialog.

Figure 20.2 The Listener tab of the Listener Preferences dialog



The Listener tab allows you to set the size of the stack used in the Listener's evaluation process. By default, this process has a stack of size determined by the value of the variable **system:*sg-default-size***. If you find you are getting stack overflow errors in correct code that you enter at the Listener prompt, then increase the stack size. This setting takes effect for subsequently created Listener windows and LispWorks sessions.

The other configurable aspects of the Listener are shared with the Editor tool, including:

- emulation, including key input and cursor styles
- display of the toolbar
- font
- the text styles used to highlight selected text, color code syntax, and so on

To alter these, raise the Listener Preferences dialog and select the appropriate tab. See "Setting Editor preferences" on page 155 for a description of these options.

20.11 Running Editor forms in the Listener

Suppose that you have code displayed in an Editor tool and you want a convenient way to run it in the Listener. Perhaps you need to capture the return value, or perhaps ou want to test several variants by evaluating edited versions of that code. The editor command **Evaluate Last Form in Listener** is useful in these cases.

20.12 Help with editing in the Listener

Two help commands are available to provide you with more information about editor commands which can be used in the Listener.

Choose **Help > Editing > Key to Command** and type a key sequence to display a description of the extended editor command it is bound to, if any.

Choose **Help > Editing > Command to Key** and supply an extended editor command to see the key sequence it is bound to, if any.

For more details about the keyboard commands and extended editor commands available, see Chapter 12, "The Editor".

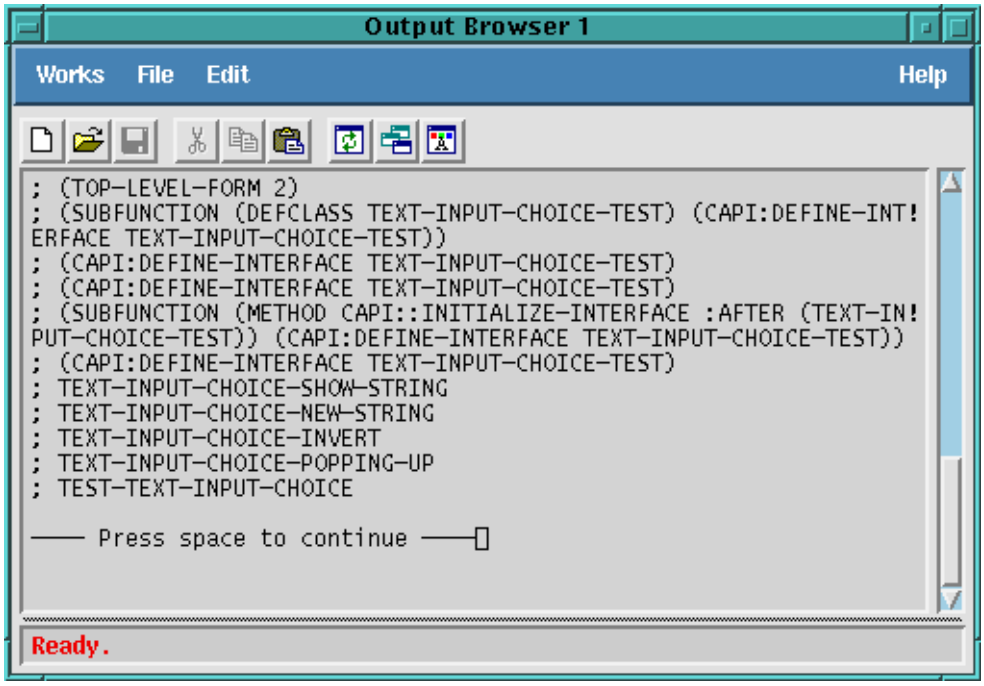
21

The Output Browser

The Output Browser is a simple tool that displays the output generated by your programs, and by operations such as macroexpansion, compilation and tracing. You can

create one by choosing **Works > Tools > Output Browser** or clicking in the Podium. Figure 21.1 shows the Output Browser.

Figure 21.1 Output browser



The Output Browser has one main area that displays the output from the environment. Output usually consists mostly of compilation, trace and macroexpansions, but can also include output from other tools, such as the Profiler. The main area is actually an editor window, so all the usual editor keyboard commands can be used in it. See Chapter 12, “The Editor” for more details about these operations.

The Output Browser is invaluable when you are developing code, because it collects any output generated by your code. An example of how to do this is given in “Viewing output” on page 11.

Many other tools in the environment contain an output view, which you can display by clicking their Output tab. The Output tab collects all the output generated by that tool. For instance, the System Browser has an Output tab that displays compilation messages. The Editor tool’s Output tab additionally displays macroexpansions. Note that

the Output Browser is the only tool which displays any output from your own code without any need for further action on your part.

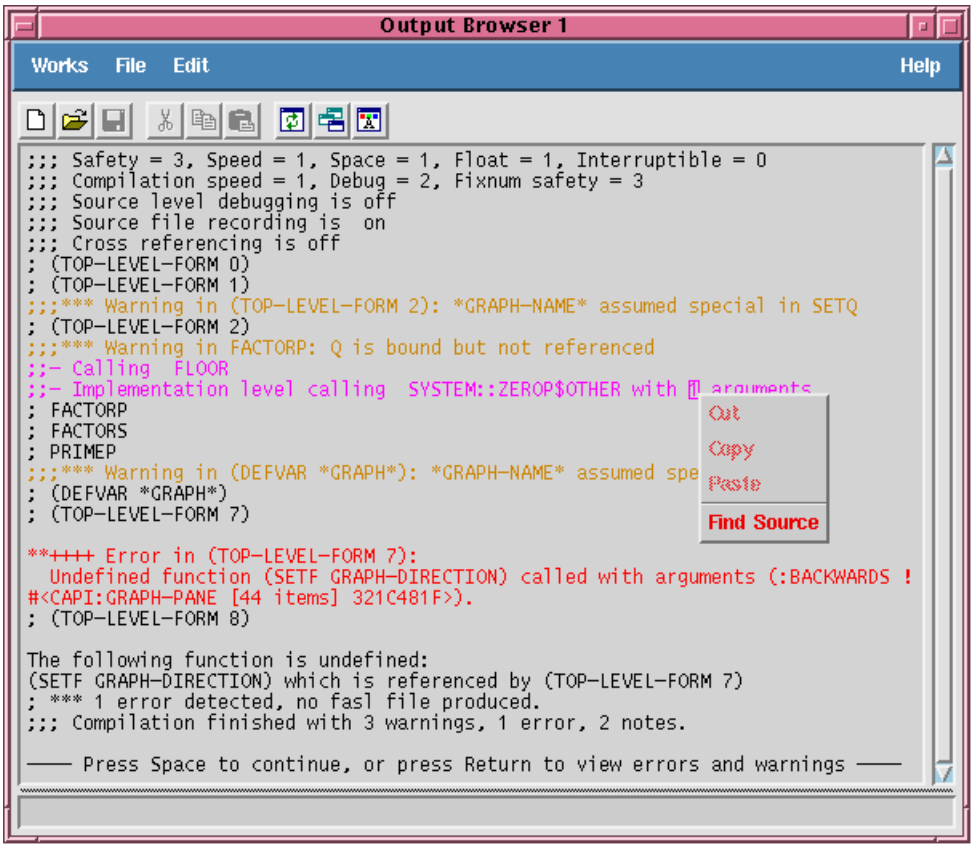
Note: The Output Browser (and the Output tab of some tools) displays only the output from the Common LispWorks environment. By default, processes not associated with the tools write their output to the terminal stream.

21.1 Interactive compilation messages

Compilation messages are highlighted in the output, with errors, warnings and optimization hints each displayed in a unique style. When the editor's cursor is within a com-

pilation message, choose **Find Source** in the context menu to display the source code where the condition occurred, in an Editor tool.

Figure 21.2 Compilation messages and the use of the context menu



You can also raise a Compilation Conditions Browser tool to view the errors and warnings directly from the output view, by pressing **Return** as mentioned in the output shown above.

Another way to visit the source code where the condition occurred is the editor's **Edit Recognized Source** command which is bound to **Ctrl+X ,** (comma) in Emacs emulation.

21.1.1 Compilation message styles

The text styles used to highlight compilation messages in the output have these meanings and default appearance:

Table 21.1 Compilation message styles

Style Name	Use	Default appearance
Compiler Note	Optimization hints	`:magenta` foreground
Compiler Warning	Warnings and other messages	`:orange3` foreground
Compiler Error	Errors	`:red` foreground

Compiler explanations are optimization hints generated by compiling code with the `:explain` declaration. See the *LispWorks User Guide* for a description of the `:explain` declaration.

Note: You can changes the styles used to display compilation messages via **Preferences... > Styles**.

22

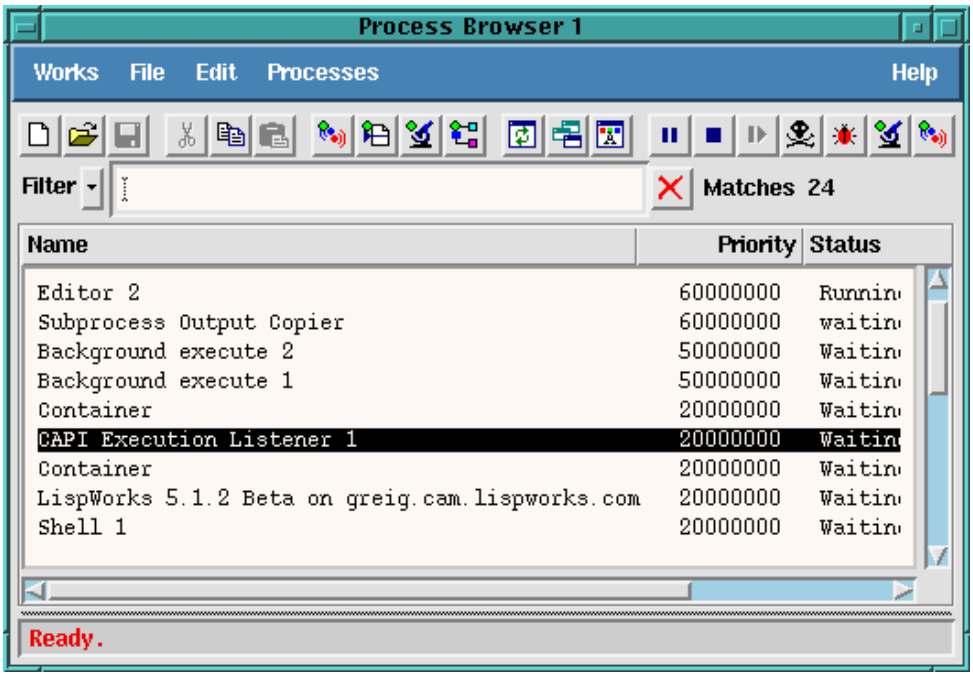
The Process Browser

The Process Browser allows you to view and control the processes in the LispWorks multiprocessing model. See the *LispWorks User Guide* for more information about multiprocessing.

Note: Each individual window in the Lisp environment runs as a distinct process in the LispWorks multiprocessing model. The whole of LispWorks runs in a single system process. On Linux and FreeBSD each LispWorks process corresponds to a single system thread.

To create a Process Browser, choose **Works > Tools > Process Browser** or click in the Podium.

Figure 22.1 .The Process Browser

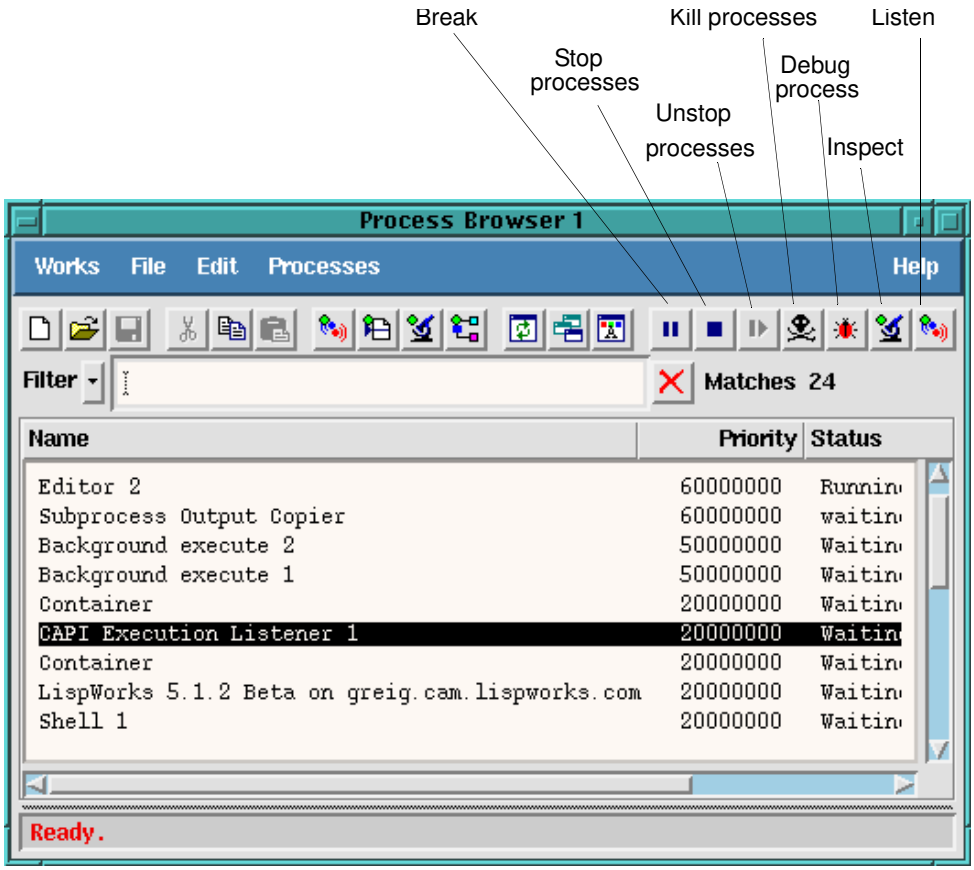


The Process Browser consists of a main area in which all the current processes in the environment are listed, and a Filter area which you can use to restrict the information displayed in the main area.

Like other filter areas, you can enter a string or a regular expression in the Filter to limit the display to only those items which match your input, or the complement of

this. See “Filtering information” on page 42 for more information about using the Filter area.

Figure 22.2 Process browser



Notice the tool bar buttons in Figure 22.2: these buttons provide the same actions as the **Process** context menu: Break, Stop, Unstop, Kill, Debug, Inspect and Listen.

22.1 The process list

The main area contains a list of all the current processes in the Lisp image. Properties of each process are shown in the columns **Name**, **Priority** and **Status**.

If you have many processes running, you can use the filter area to only list processes containing a given string. For example, if you enter “Running” in the filter area, and click on ✓ then only processes that have the word “Running” in their description will be shown.

The processes displayed in the main area can be sorted by clicking the relevant button above each column. For example, to sort all listed processes by process priority, click on the **Priority** title button.

22.2 Process control

The **Processes** menu contains commands that let you control the execution of processes in the Lisp image. These same commands are available using the toolbar buttons at the top of the Process Browser window or by using the context menu (Use the left mouse button to select a process; the context menu is usually accessed by the right mouse button.) Process commands act on the process that has been selected in the process list. You can select a process by clicking on the line in the process list that contains the process name and status information.

Choose **Processes > Break** to break the current process. This breaks Lisp and gives you the opportunity to follow any of the normal debugger restarts.

Choose **Processes > Kill** to kill the selected process.

Choose **Processes > Stop** to stop the selected process. The process can be started again by choosing **Processes > Unstop**, and thus is similar to the use of **`Ctrl+Z`** in a UNIX session.

Choose **Processes > Unstop** to restart a process which has been stopped using **Processes > Stop**. This is similar to the use of the UNIX command **`fg`**.

Choose **Process > Inspect** to call up a Common LispWorks Inspector tool to inspect the selected process. See Chapter 16, “The Inspector” for more information on inspecting objects and processes.

Choose **Process > Listen** to make the selected process be the value of **`*`** in a Listener tool. See Chapter 20, “The Listener” for more information on using the Listener tool.

Choose **Processes > Remote Debug** to debug the current process in a Debugger tool.See Chapter 10, “The Debugger Tool” for more information on using the Debugger tool.

Note: do not break, kill, stop or remote debug system processes. This may make your environment unusable.

Note: you cannot control the GC monitor from the Process Browser, since this runs as a separate UNIX process.

22.3 Other ways of breaking processes

If you do not wish to use the Process Browser to break a particular process, you can use the keyboard command **`Alt+Ctrl+C`** to break the current process. Note that you must use the true Alt key on your keyboard for this command. Alternatively, click the **Interrupt Lisp** button on the GC Monitor window.

In the Listener tool, you can break the evaluation process as described in "Interrupting evaluation" on page 295.

You can break a process by calling the function **`mp:process-break`**.

22.4 Updating the Process Browser

The Process Browser updates itself automatically when a new process is created and when a process terminates.

In the initial configuration the Process Browser does not automatically update on any other event, so changes such as processes sleeping and waking are not noticed immediately. There are two ways to ensure such changes are visible in the Process Browser:

- You can do **Works > Refresh** to view the latest status displayed for each process, or
- The Process Browser can be made to update automatically, as described in "Process Browser Preferences" on page 315.

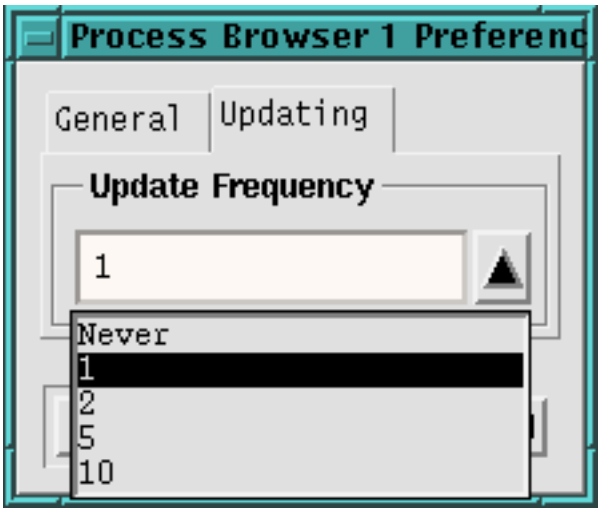
22.5 Process Browser Preferences

Choose **Works > Tools > Preferences...** or click to display the Process Browser preferences dialog.

You can control whether the Process Browser displays the process operations toolbar by the option **Show Toolbar**, as described in "Toolbar configurations" on page 24.

You can make the Process Browser update automatically at a predetermined frequency by setting the option **Update Frequency**, as illustrated in Figure 22.3. The update periods are in seconds.

Figure 22.3 Configuring the Process Browser to update automatically



23

The Profiler

23.1 Introduction

The Profiler provides a way of monitoring Lisp functions during the execution of your code. It is likely that you can make your code more efficient using the data that the Profiler displays.

The Profiler helps you to identify functions which are called frequently or are particularly slow. You should concentrate your optimization efforts on these routines.

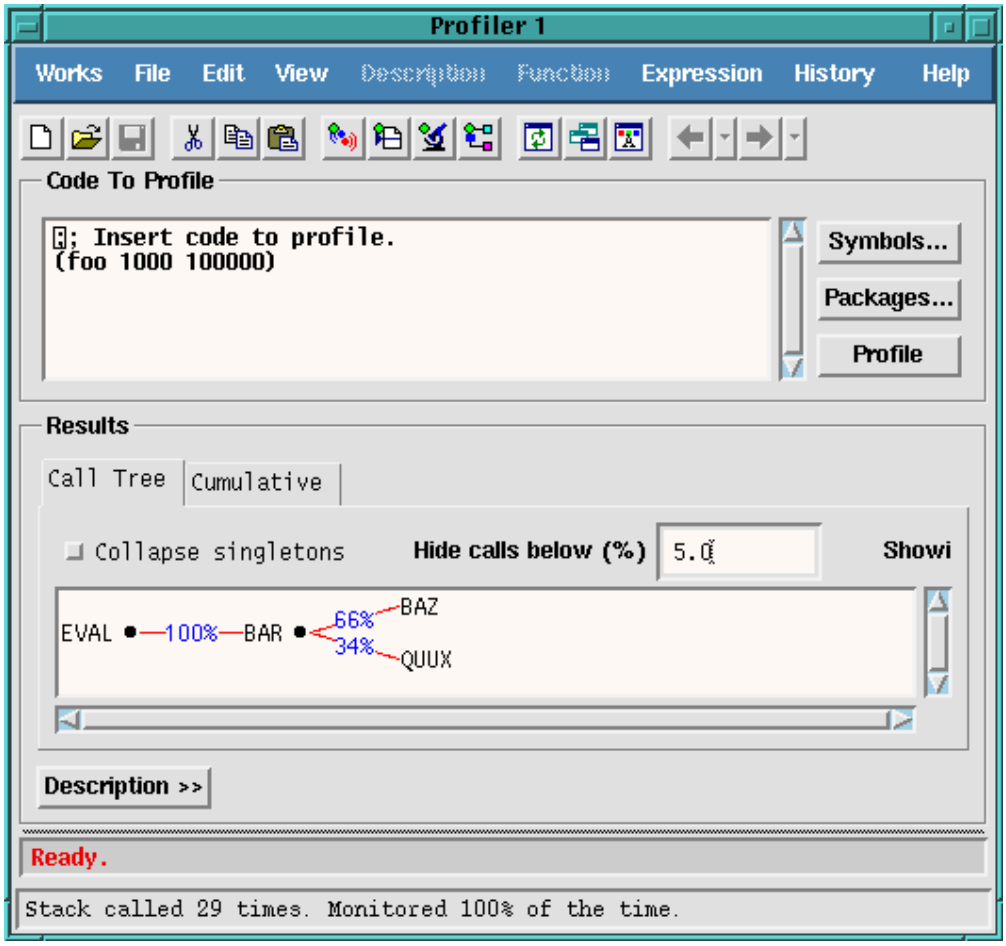
The Profiler gives you an easy way of choosing which functions you wish to profile, which code you want to run while profiling, and provides you with a straightforward display of the results of each profile.

To create a Profiler, choose **Works > Tools > Profiler** or click in the Podium.

In the next two sections we assume you are profiling a call to the function `foo` defined as follows:

```
(in-package "CL-USER")

(defun baz (l)
  (dotimes (i l)))

(defun quux (l)
  (dotimes (i l)))

(defun bar (n l)
  (dotimes (i n)
    (baz l))
  (dotimes (i n)
    (quux (floor l 2))))

(defun foo (n l)
  (bar n l))
```

Figure 23.1 The Profiler



The Profiler has five areas:

The Code to Profile panel lets you set up and profile any amount of Lisp source code.

- Use the large text box to enter the Lisp source code that you wish to profile. This text area is actually an editor window, similar to those described in “Basic Editor commands” on page 164.
- The **Symbols...** and **Packages...** buttons let you choose which symbols and packages you want to profile.

- The **Profile** button runs and profiles the source code you entered.

The Results area is used to display the results of a profile, described in detail in “Display of Profiler Data” on page 320.

The Description area optionally shows a description of a function in the profile data. You can show the description by clicking on the **Description >>** button. The name, function object, lambda list, documentation string and source files of the selected function are displayed. The context menu in the description area allows further operations. Hide the description area if you wish by clicking on the **Description <<** button.

The Echo area allows interaction with editor commands, as in other tools.

The bottom pane of the Profiler displays a summary of the last profile performed.

23.2 Display of Profiler Data

There are two tabs in the results area of the Profiler. These tabs display two sorts of data collected during profiling.

23.2.1 Call Tree

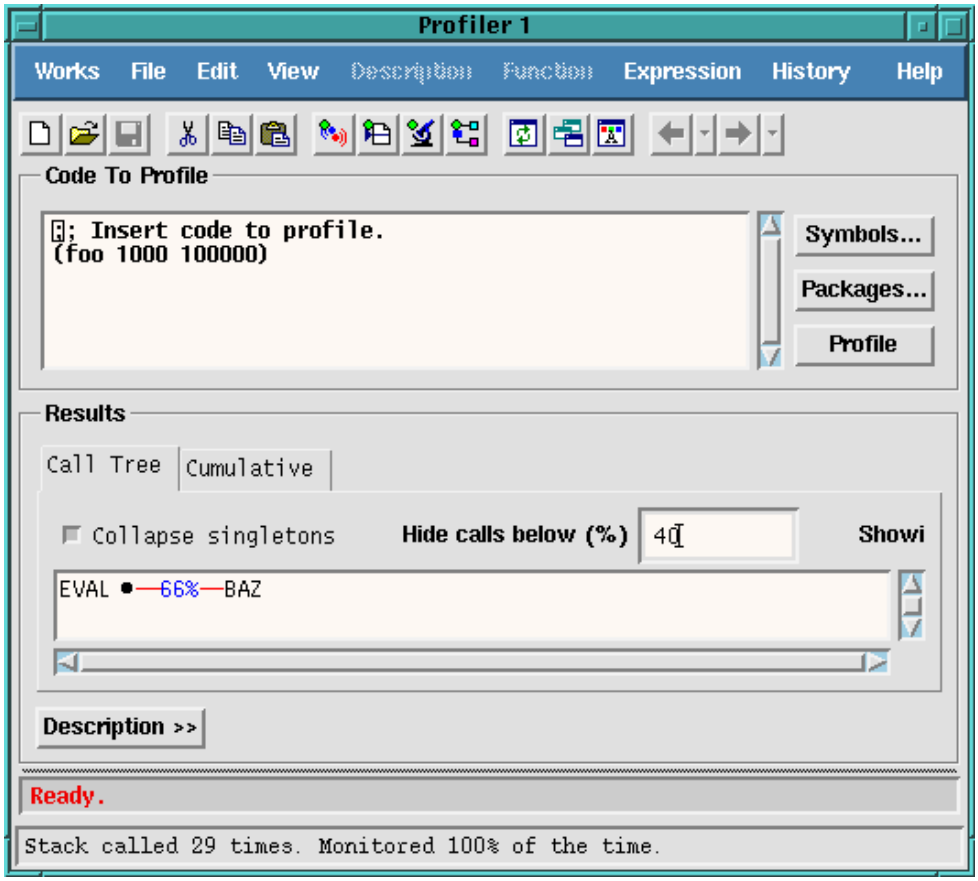
The call tree tab shows a graph of functions called by the top level function call that was profiled. Each node represents a function call. The graph edges are labelled according to the proportion of time spent in each function call. For example in Figure 23.1, of all the time spent in function **`bar`**, most was spent in **`baz`** and the rest in **`quux`**. This allows you to see which branches of the code dominate the total time spent.

When optimizing your code you will want to concentrate on the calls which take a large proportion of the time. The least significant parts of the graph are removed from the display according to the percentage in the "Hide calls below (%)" box. You can adjust this percentage simply by entering an integer and pressing **`Return`**.

When analysing the call tree to find the most significant branches, single callees (that is, functions which account for all of the time spent by their caller) are not interesting.

You can adjust the call tree to omit these functions from the displayed graph by checking the "Collapse singletons" box.

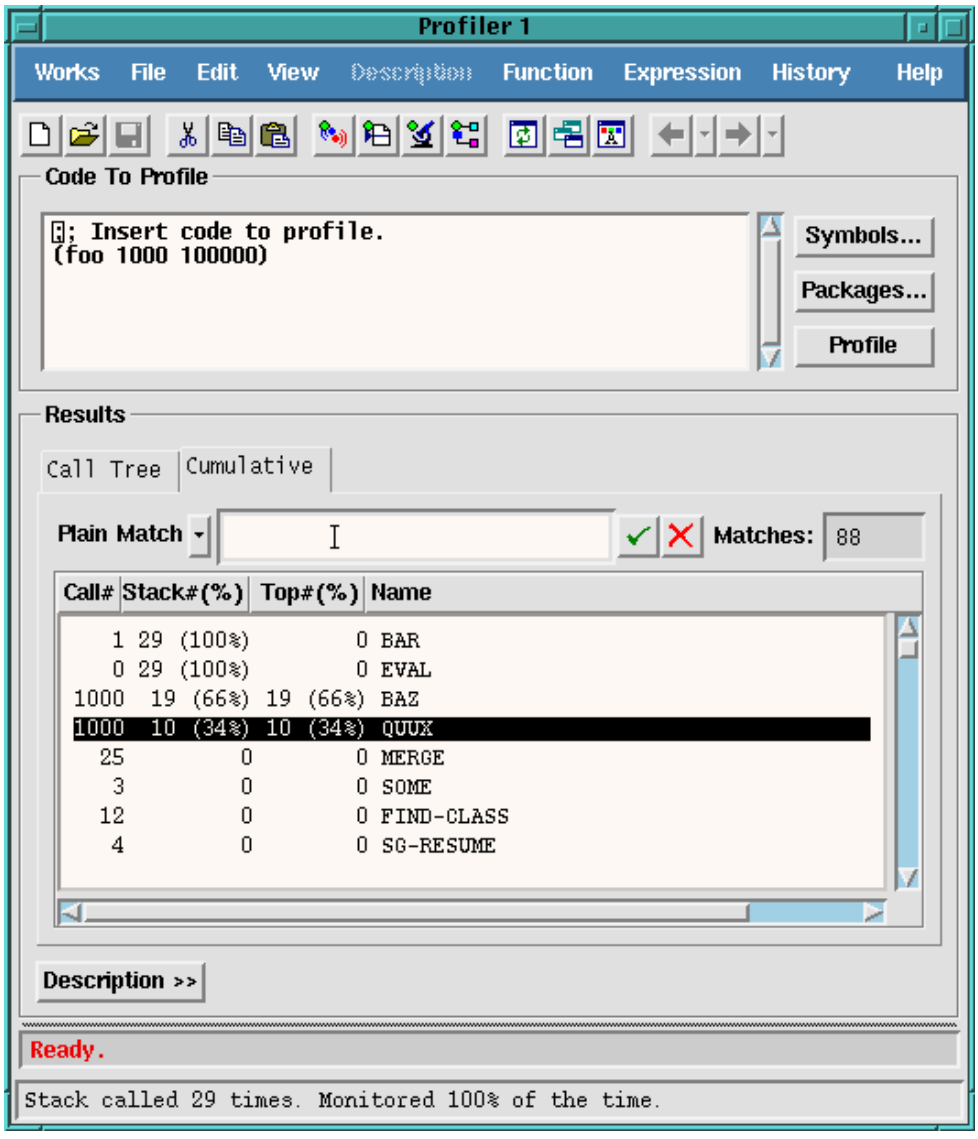
Figure 23.2 The Profiler's Call Tree adjusted



You can also change the root of the graph displayed. This option is useful when you have a large call tree. To do this, select the node that you wish to make the root, and raise the context menu, and choose the **Set As Root** command. To display the entire call tree again, choose the **Show Whole Tree** command from the graph's context menu.

23.2.2 Cumulative Results

Figure 23.3 The Profiler's Cumulative Results view



- The main list displays the functions called during the last profile. For each function, the list displays the number of calls, the number of times it was found on the stack, and the number of times it was found at the top of the stack.
- The Filter box lets you restrict the display of information in the Results area.

23.3 A description of profiling

When code is being profiled, the Lisp process is interrupted regularly at a specified time interval. At each interruption, the Profiler scans the execution stack and records the name of every function found, including a note of the function at the top of the stack. Moreover, a snapshot of the stack is recorded at each interruption, so we know not merely how many times we reach a function call, but also how we reached that call.

When profiling stops (that is, when the code being profiled has stopped execution) the Profiler presents the data in two tabs.

23.3.1 Description of call tree data

The call tree is collated from all the stack snapshots taken during profiling. The Call Tree tab shows a graph in which:

- The graph nodes represent function calls
- The graph edges show the proportion of time spent in each call.

Each parent node represents the caller function, so that the ancestors of each leaf node represent the entire stack. The graph edges are labelled with the time spent in the child call as a percentage of the time spent in the parent call, these times averaged over all the profile data collected.

23.3.2 Description of cumulative data

The Cumulative tab shows aggregated information about each function that includes the following information:

- The number of times each function being profiled was called.
- The number of times each 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 each 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.

23.4 Steps involved in profiling code

Each time you profile code, you first need to set up the profiler to ensure that you find out the sort of information you are interested in. This section gives you details about how to go about this.

The steps that you need to take when profiling code are as follows:

1. Choose which functions you want to profile.
2. Specify the code that you want to run while profiling.
3. Perform the profile.

Note: You do not have to adhere strictly to the sequence shown above, but this is the order that you should usually follow.

See the chapter on the Profiler in the *LispWorks User Guide* for information on advanced configuration of the profiler.

23.4.1 Choosing the functions to profile

It is possible to keep track of every function called when running code, but this involves significant effort in determining which functions are suitable for profiling and in keeping track of the results. To minimize this effort you should specify which functions you want to profile. The profiler checks that these functions have indeed got function definitions and are therefore suitable for profiling. For more information on the types of function that can be profiled, see “Profiling pitfalls” on page 330.

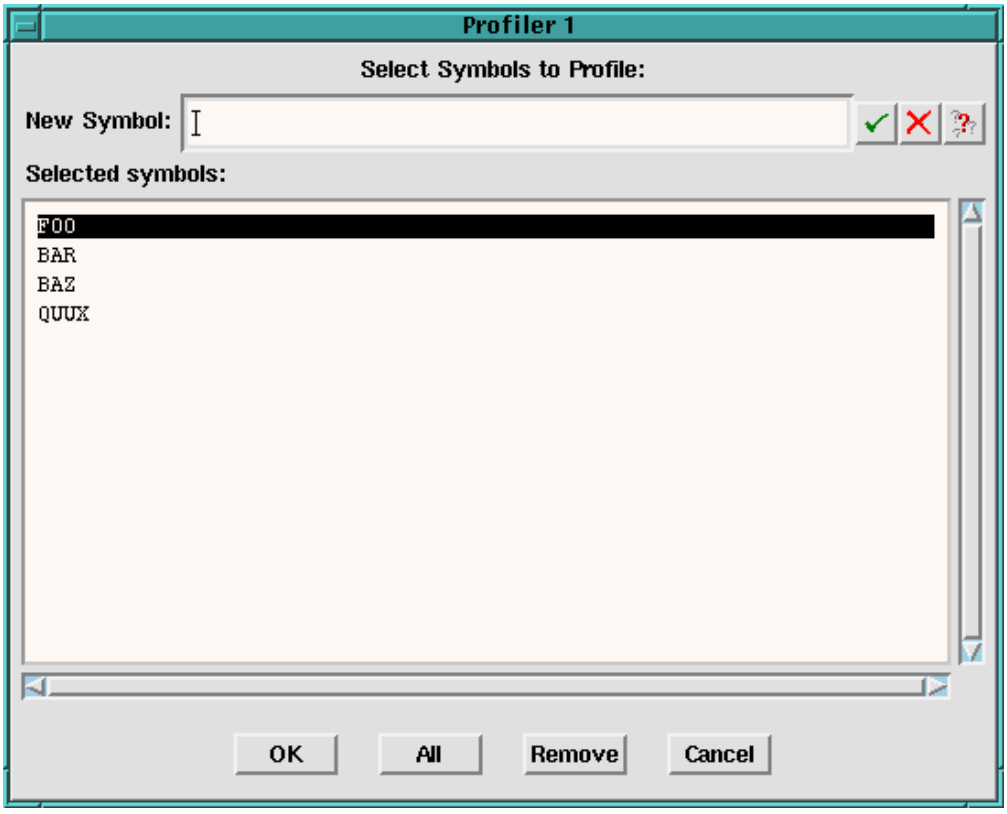
There are two ways of specifying functions that you want to profile:

- Choose which individual functions you want to profile.
- Choose whole packages, all of whose functions are profiled.

23.4.1.1 Choosing individual functions

Click **Symbols...** to specify a list of Lisp functions that you want to profile. The dialog shown in Figure 23.4 appears.

Figure 23.4 Select Symbols to Profile dialog



This dialog displays the list of functions to be profiled.

- To add a function to the list, enter its name in the **New Symbol** text box and click ✓.
- To remove a function from the list, select it from the list and click **Remove**.
- To remove several functions, select them all before clicking **Remove**.

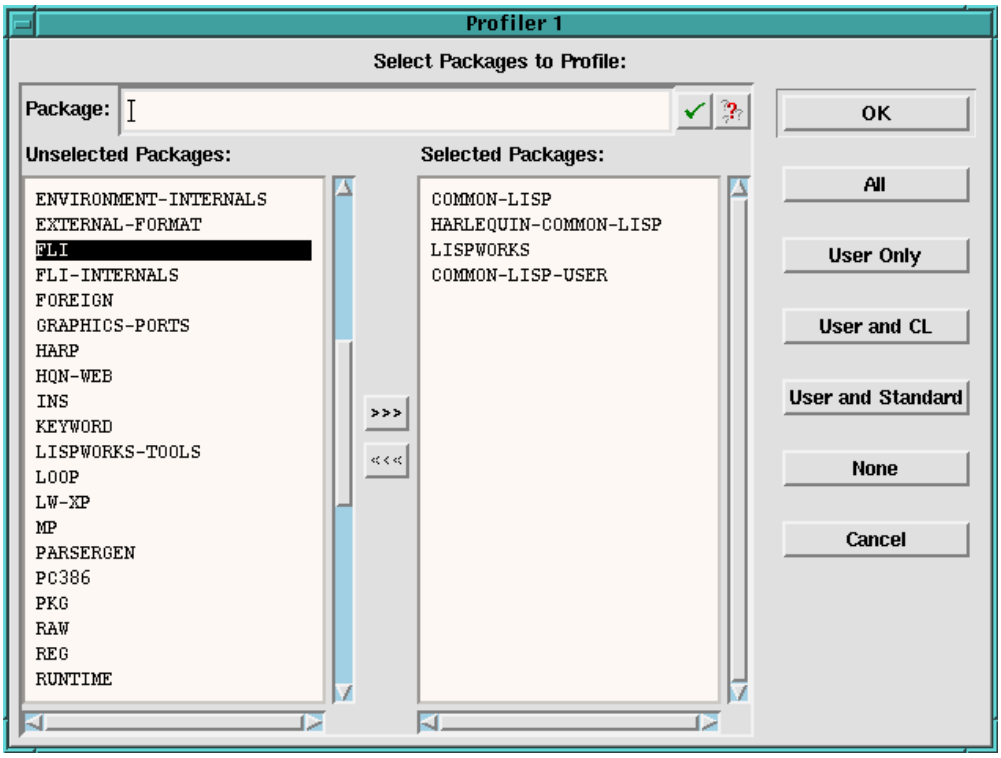
Click **OK** when you have finished choosing symbols.

Note: whilst entering the function name in the **New Symbol** text box you can click click to use completion. This allows you to select from a list of all symbol names which begin with the partial input you have entered. See "Completion" on page 46 for detailed instructions.

23.4.1.2 Choosing packages

You may often want to profile every function in a package. Click **Packages...** to specify a list of packages whose functions you want to profile. The dialog shown in Figure 23.5 appears.

Figure 23.5 Select Packages to Profile dialog



The main part of this dialog consists of two lists:

- The Unselected Packages list shows packages in the Lisp image whose functions are not to be profiled.

- The Selected Packages list shows packages in the Lisp image whose functions are to be profiled.

A global function will be profiled if its symbol is visible in one of the Selected Packages.

To modify the Selected Packages list:

1. Consider whether one of these buttons offers what you need, or close to it:

   All Adds all packages.

   Note: There are significant processing overheads when profiling all functions in all packages, and the results you get may include much unwanted information.

   User Only Adds the CL-USER package.

   User and CL Adds the CL-USER and CL packages.

   User and Standard

   Adds the CL-USER package along with those packages that you have defined.

   Note: The Profiler tool assumes that packages not named in the value of **`*packages-for-warn-on-redefinition*`** are user-defined.

2. Add to your Selected Packages list if necessary. You can add a single package in one of three ways:

- Type the package name in the Select Package box and press **`Return`** or click ✓, or
- Select the package in the Unselected Packages list and click on the **`>>>`** button, or
- Double-click on the package in the Unselected Packages list.

3. Remove packages from the Selected Packages list if necessary. You can remove a single package in one of two ways:

- Select the package in the Selected Packages list and click on the **`<<<`** button, or
- Double-click on the package in the Selected Packages list.

Also you can click the **None** button to clear the list of Selected Packages. This is useful is you only want to profile a few functions, which you can specify easily using the **Symbols...** button on the Profiler tool itself.

4. Finally, click **OK** to dismiss the dialog when you have finished selecting the packages whose functions you want to profile, or click **Cancel** to cancel the operation. This also dismisses the dialog.

23.4.2 Choosing the time interval

See the chapter on the profiler in the *LispWorks User Guide* for information on how to set the time interval at which you want the Lisp process to be interrupted.

23.4.3 Specifying the code to run while profiling

Code which is to be executed during profiling should be entered in the Code to Profile area. This is actually an editor window, and so you can use all the keyboard commands which can be used in the editor.

Code may be placed in this window in three ways:

- Type it directly into the window
- Paste it in from other editor windows in the environment
- Paste it in from other applications

Specify the package in which you want to run the code to be profiled using the Package box in the General tab of the Profiler Preferences dialog. Choose **Works > Tools > Preferences...** or click to display this dialog. If you are unsure, full details on how to do this can be found in "Specifying a package" on page 34. Like all other tools in the environment, the Profiler can have a particular package associated with it; the default package is **CL-USER**.

23.4.4 Performing the profile

Once you have set up the profile as described above, perform the profile itself by clicking on the **Profile** button in the Profiler.

23.5 Format of the cumulative results

After you have run the profile, a four column table is printed in the large list in the Cumulative tab of the Results area. These columns are laid out as follows:

Call#	The call count of each function, that is, the number of times it was called during execution of the code.
Stack#(%)	The number of times the function was found on the stack when the Lisp process was interrupted. The parenthesized figure shows the percentage of time the function was found on the stack.
Top#(%)	The number of times the function was found on the top of the stack when the Lisp process was interrupted. Again, the figure in brackets shows the percentage of time the function was found on top of the stack.
Name	The name of the function.

You can order the items in the list by clicking on the relevant heading button.

Selecting any item in the list displays a description of that function in the Description area. In addition, an item selected in the main list can be acted upon by any relevant commands in the **Function** menu. For instance, if you select a generic function in the main list and choose **Function > Generic Function**, you can view the generic function in a Generic Function Browser. This is consistent with many of the other tools in the environment.

Double-clicking on an item in the Description list invokes an Inspector on the selected item. In addition, an item selected in this area may be acted on by any relevant commands in the **Description** menu, as is the case with many other tools in the environment. For instance, choose **Description > Copy** to copy the item selected in the Description list to the clipboard. See “Performing operations on selected objects” on page 35 for details on the commands available.

23.6 Interpreting the cumulative results

The most important columns in the Cumulative tab are those showing call count (call#) and number of times on the top of the stack (Stack#). Looking solely at the

number of times a function is found on the stack (Stack#) can be misleading, because functions which are on the stack are not necessarily using up much processing time. However, functions which are consistently found on the top of the stack are likely to have a significant execution time. Similarly the functions that are called most often are likely to have the most significant effect on the program as a whole.

23.7 Profiling pitfalls

It is generally only worth profiling code which has been compiled. If you profile interpreted code, the interpreter itself is profiled, and this skews the results for the actual Lisp program.

Macros cannot be profiled because they are expanded during the compilation process.

23.7.1 Effects of random sampling

Always bear in mind that the numbers produced are from random samples, so you should be careful when interpreting their meaning. The rate of sampling is always coarse in comparison to the function call rate, 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. In practice, however, this is not usually a problem.

23.7.2 Recursive functions

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, and so the total number of times a function is found on the stack may be greater than the number of times the stack is examined.

23.7.3 Structure accessors

You must take care when profiling structure accessors. These 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.

23.7.4 Consequences of restricted profiling

Even if you configure the Profiler to profile all the known functions of an application, it is possible that less than 100% of the time is spent monitoring the top function. This is because an internal system function could be on the top of the stack at the time of the interrupt.

If you configure the Profiler to omit certain functions then of these will not be displayed in the Results area, and so the display may not match what your source code.

23.7.5 Effect of compiler optimizations

With certain compiler settings code can be optimized such that the Profiler data does not appear to match your source code. For example when a tail call is optimized, the tail-called function appears in the call tree as a child of the parent of the caller, rather than as a child of its caller (just as in the debugger stack). Similarly code using **`funcall`** or **`apply`** may yield confusing results. To prevent tail-call optimization, use compiler setting **`debug 3`**.

23.7.6 Effect of compiler transforms

The compiler may transform some functions such that they are present in the source code but not in the compiled code.

For example, the compiler transforms this source expression:

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

into this compiled expression:

```
(memq 'x '(x y z))
```

Therefore function **`memq`** will appear instead of **`member`** in the profile results.

Similarly, you cannot profile inlined functions.

23.8 Some examples

The examples below demonstrate different ways in which the profiler can be configured and code profiled so as to produce different sets of results. In each example, the following piece of code is profiled:

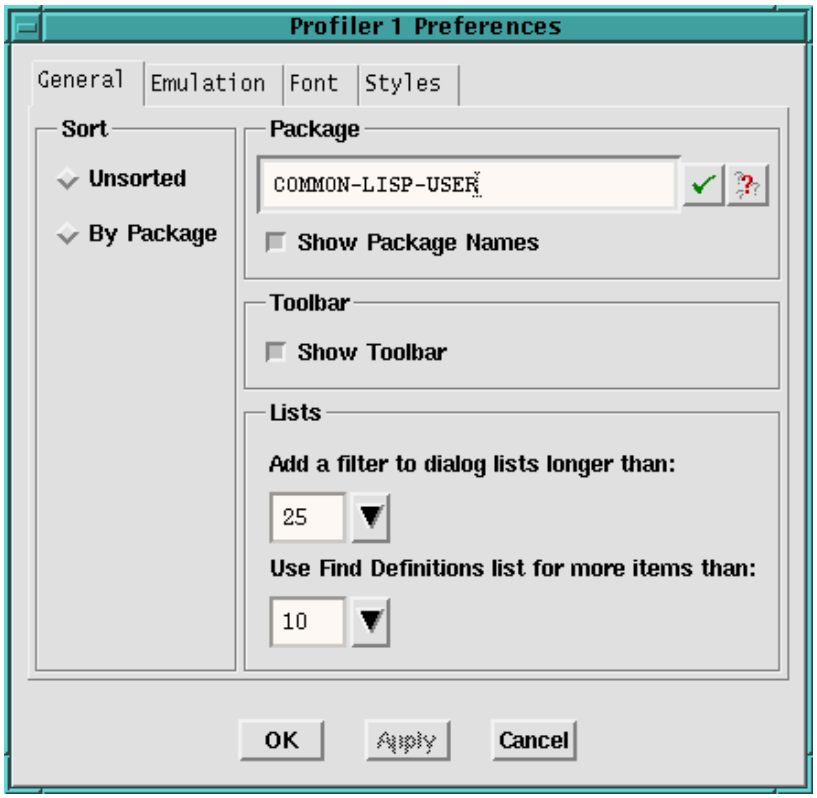
```
(dotimes (x 1000)
  (capi:make-container
   (make-instance 'capi:title-pane
                  :text "Title")))
```

This is a simple piece of code which makes some CAPI objects.

1. Create a Profiler tool if you have not already done so.
2. Copy the code above into the box in the **Code to Profile** panel.

3. Choose **Works > Tools > Preferences...** or click in the Profiler, and select the General tab, to change the package of the Profiler.

Figure 23.6 Profiler Preferences dialog



4. In the Profiler Preferences dialog, replace the default package in the Package text box with **CAPI** and click ✓.
5. Click **OK** to dismiss the Preferences dialog and apply the change you have made.
6. Click on **Profile**.

This profiles the functions in the **COMMON-LISP**, **CL-USER** and **LISPWORKS** packages.

Next, add the **CAPI** package to the list of packages whose functions are profiled.

7. Click **Packages**.

8. In the dialog, double-click on **CAPI** in the **Unselected Packages** list, and click on **OK**.
9. Click on **Profile** to profile the code again.

   Notice that this time there are many more functions which appear in the profile results.
10. Select a few of the functions listed at the top of the **Results** area, and look at their function descriptions.

    Add the Description area by clicking the **Description >>** button if you have not already done so.

    Notice that most of the functions appearing on the stack are in the **CAPI** package. It is worth profiling a few functions explicitly, and removing unwanted packages from the list of packages to profile.
11. Click **Symbols...**, and add the following four functions to the list in the dialog:

    ```
    merge find-class make-char functionp
    ```

    Type the name of each function and press **Return** or click ✓ to add it to the list.
12. Click **OK** when you have finished adding to this list.

    Now remove the unwanted packages from the list of packages to profile, as follows:
13. Click **Packages...**.
14. In the dialog click on **None** to remove all items in the Selected Packages list
15. Click on **OK**, and profile the code again by clicking on **Profile**.

Notice that the four functions in the **COMMON-LISP** package are still being profiled, even though you are no longer profiling all functions from that package by default.

24

The Shell and Remote Shell Tools

24.1 Introduction

You can run a UNIX command line session from within Common LispWorks by using the Shell tool. The Shell tool automatically runs on your current host.

Also available is a Remote Shell tool which runs a session on another UNIX machine on your network.

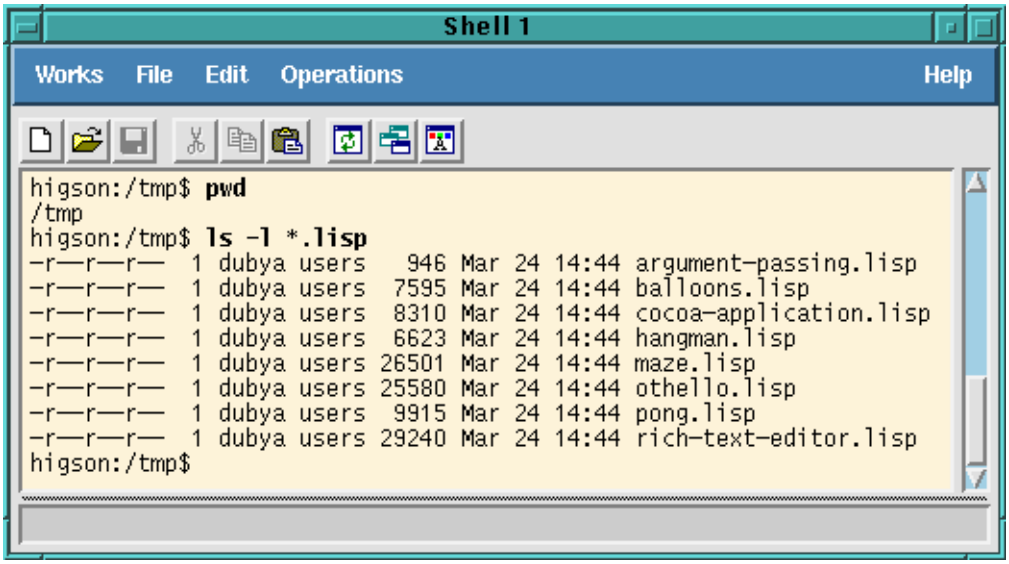
24.2 The Shell tool

You can create a Shell tool in one of two ways:

- Choose **Works > Tools > Shell** or click in the Podium.

- Type the extended command **`Meta+X Shell`** in any Editor window (or any other window based on an editor, such as the Listener).

Figure 24.1 The Shell tool



The **Operations** menu contains the following commands which send UNIX signals to the shell process. These only work on UNIX/Linux/FreeBSD/Mac OS X.

Choose **Operations > Interrupt** to send a break signal to the shell process. This stops the current task and returns control to the UNIX command line in the Shell tool, if necessary.

Choose **Operations > Suspend** to send a suspend signal to the shell process. This suspends the current task so that you can continue entering commands at the UNIX command line. To resume the task, type **`fg`** at the UNIX command line in the Shell tool. Alternatively, type **`bg`** at the command line to force a task to run in the background.

Choose **Operations > Eof** to send an EOF signal to the process.

24.3 Command history in the shell

The Shell tool is another example of a tool which is based on an editor, and thus many of the keys available in the editor are also available in the Shell tool.

Like the Listener, the Shell tool is run in execute mode, which means that several additional keystrokes are available in EMACS emulation, as follows:

Press **Meta+P** or **Ctrl+C Ctrl+P** to display the previous command entered in the shell.

Press **Meta+N** or **Ctrl+C Ctrl+N** to display the next command in the history.

Press **Meta+R** or **Ctrl+C Ctrl+R** to perform a search of the command history.

24.4 Configuring the shell to run

This section applies only on UNIX/Linux/FreeBSD/Mac OS X.

By default, the Shell tool runs the UNIX command shell known as **bash**. If you would rather use a different shell (such as **csh**, **tcsh**, **ksh**, etc.), or if you do not have **bash** available on your UNIX system, then change the value of the variable **editor:*shell-shell***, which has the value **nil** by default. This means that the Shell tool will use the value of the variable **ESHELL** or **SHELL** if set, or one of "/bin/sh" (system V) and "/bin/csh" (otherwise).

24.5 The Remote Shell tool

This looks similar to the Shell Tool, but you must specify which host to run the remote shell on when you start it up.

To start a Remote Shell tool, enter **Meta+X Remote Shell** in the Editor or Listener tool, and supply the hostname of the remote machine when prompted.

The tool runs an appropriate UNIX command (**rsh** or **remsh**) with the hostname which you specify.

25

The Stepper

25.1 Introduction

The Stepper tool allows you to follow the execution of your program, displaying the source code as it executes. While *stepping*, you can see the evaluation of each subform, function call and the arguments and return values in each call. At every call to one of your functions, you have the option of stepping into that function, that is stepping the source code definition of the function.

Where a macro appears in stepped code, the Stepper can macroexpand the form and step the resulting expansion, or simply step the visible inner forms of the macro form. Where a special form such as **`if`** appears in stepped code, the Stepper processes it according to the execution order in that special form.

The system creates a Stepper tool automatically when your code reaches a breakpoint

Other ways to start a Stepper tool are:

- Choose **Works > Tools > Stepper** or press in the Podium and enter a single form

- Choose **Frame > Restart Frame Stepping** in a Debugger tool

Figure 25.1 The Stepper

The Stepper has four areas:

25.1.1 Stepper toolbar

The commands on the stepper toolbar allow you to step to various points in the code, set breakpoints and perform macroexpansions.

25.1.2 Source area

This is an editor window where you should enter the initial form to step.

When you step into a function, a read-only copy of its definition is shown in the source area.

Notice that the editor cursor is an underline in the source area: this is because the normal cursor styles are not visible where the Stepper is highlighting a form.

25.1.3 Backtrace area

The Backtrace tab displays the function calls on the execution stack in the code being stepped.

The topmost item in the backtrace area shows the next step, known as the *status*. When calling a function, the status item contains the arguments and when returning from any form, the status item contains its return values. You can see the contents of the status item by expanding it. You can make the status item expand automatically if you wish, as described in "Backtrace preferences" on page 360.

The second topmost item in the backtrace area is the *active frame*. This shows the function executing when the breakpoint was reached, and its arguments.

For function calls, arguments and local variables can be seen by expanding the item. You can make the active frame expand automatically if you wish, as described in "Backtrace preferences" on page 360. Just as in the Backtrace area of the Debugger tool, these stack frames and variables can be operated on using the **Frame** and **Variables** menus. For details, see "Backtrace area" on page 119.

25.1.4 Listener area

The Listener tab provides a Listener in which the execution steps are indicated. Commands can be entered here as an alternative to using the buttons on the "Stepper toolbar" on page 340.

Any form entered here is evaluated on the dynamic environment of the function being stepped.

Moreover, you can use the debugger commands such as `:v`, which prints the local variables in the current frame. You can use the value of a local variable simply by entering its name as shown. See the *LispWorks User Guide* for more details about the debugger commands.

See "Listener area" on page 357 for more details.

25.2 Simple examples

There are two ways to enter the Stepper tool:

25.2.1 Standalone use of the stepper

1. Compile and load the demo system defined in the LispWorks library file **examples/tools/demo-defsys.lisp**. First, load this file to define the system. Then evaluate in the Listener:

   ```
   (compile-system "demo" :load t)
   ```

   Note: for another way to compile and load a system, see Chapter 26, "The System Browser".

2. Create a Stepper tool by choosing **Works > Tools > Stepper** or pressing in the Podium.

3. Enter this form in the Source area of the Stepper tool:

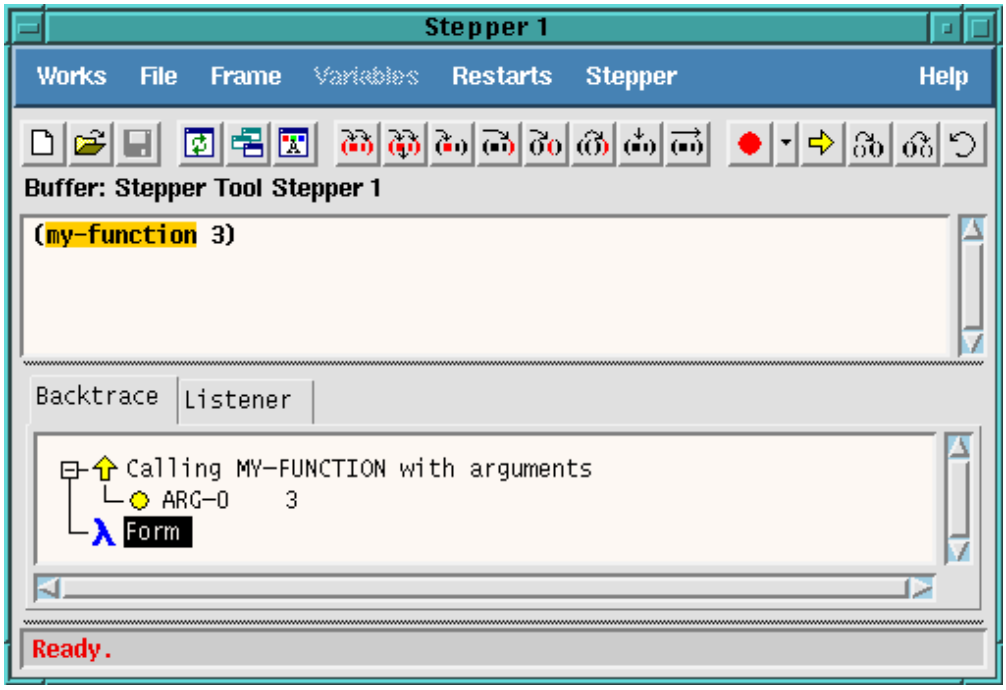
   ```
   (my-function 3)
   ```

4. Choose the menu command **Stepper > Step**. The open parenthesis is highlighted orange, indicating that the next step is to evaluate the form.

5. Choose **Stepper > Step** again. The symbol **my-function** is now highlighted orange, indicating that the next step is to call this function. Notice how the current stepping position is always highlighted orange.

6. Notice how the topmost item in the Backtrace area always indicates the next step. Expand this item to show the arguments.

Figure 25.2 Stepper backtrace showing the next step



7. At this point we have the option to step **my-function** itself, but for the moment simply choose **Stepper > Step** again, which steps to the point where the function call returns. The Backtrace area shows the return value, 12, when you expand the status item.

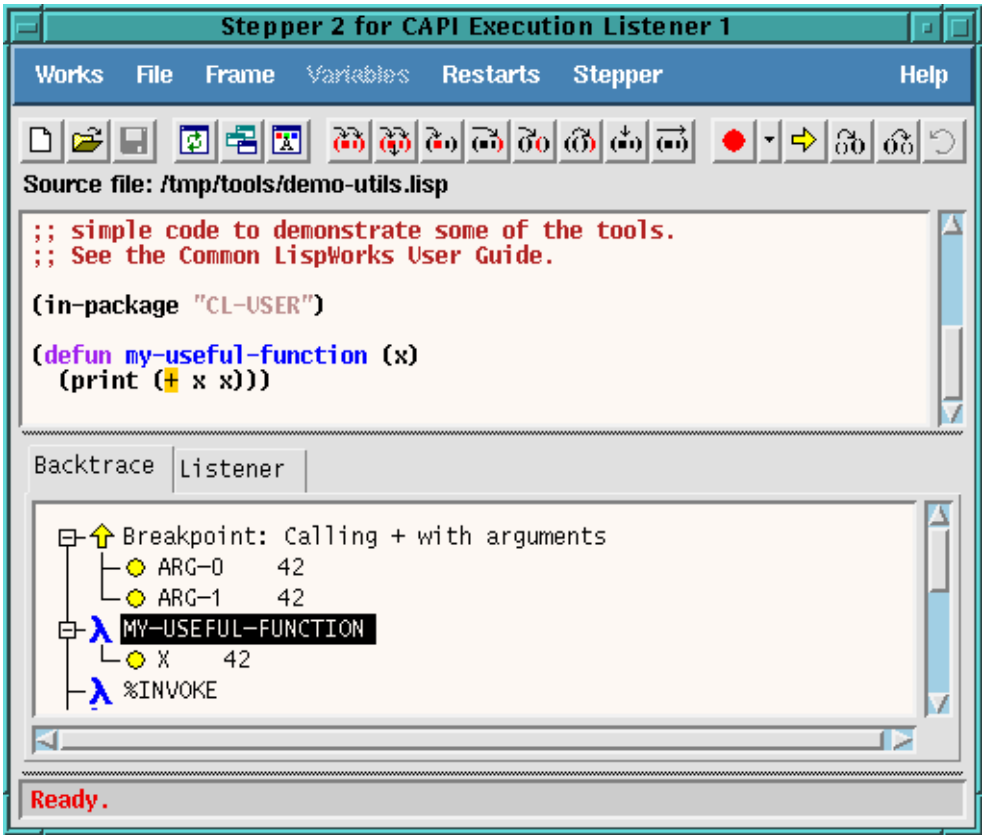
Note how the **Step** command always steps only inside the current form, and does not step into other functions.

25.2.2 Invoking the Stepper via a breakpoint

1. Compile and load the code in the system **demo** defined in the LispWorks library file **examples/tools/demo-defsys.lisp**
2. Open the file **examples/tools/demo-utils.lisp** in an Editor and set a breakpoint at the call to **+** as described in “Setting breakpoints” on page 181.

3. Evaluate `(my-useful-function 42)` in a Listener.
4. A Stepper tool appears, with the current stepping position at the breakpoint.

Figure 25.3 Stepper invoked by reaching a breakpoint



5. You can now step this code, just as in standalone mode.
6. When you choose **Stepper > Continue**, or otherwise finish stepping, `my-useful-function` returns, the Stepper is hidden and the Listener tool becomes active again.

25.3 The implementation of the Stepper

It is important to understand the following points about the implementation of the Stepper.

25.3.1 Requirements for stepping

The code you step must have been compiled, evaluated or loaded in the Lisp image.

25.3.2 Editing source code

While the Stepper is running, it displays a read-only copy of the source in the source area. Therefore, you cannot edit the code in the source area, other than when the status is "Enter a form to step in the pane above.".

If you step a function for which the source has been edited since it was compiled, then the Stepper uses a copy of the compile-time source, not the edited source.

This copy is stepped in a new editor buffer created specially for it and this is displayed in the source area.

25.3.3 Side-effects of stepping

When the Stepper steps a definition for the first time, it evaluates it.

This will not normally alter the behavior of your program, but there are three situations where this will cause unexpected behavior:

- The code is loaded from a fasl file which is not compatible with the corresponding source file.
- The source relies on compile-time side-effects of forms preceding it in the file.
- The defining form has other side effects. This is unlikely to matter for simple definers such as **`defun`** and **`defmethod`**.

25.3.4 Atomic and constant forms

It is not possible to step to atomic forms or constant forms.

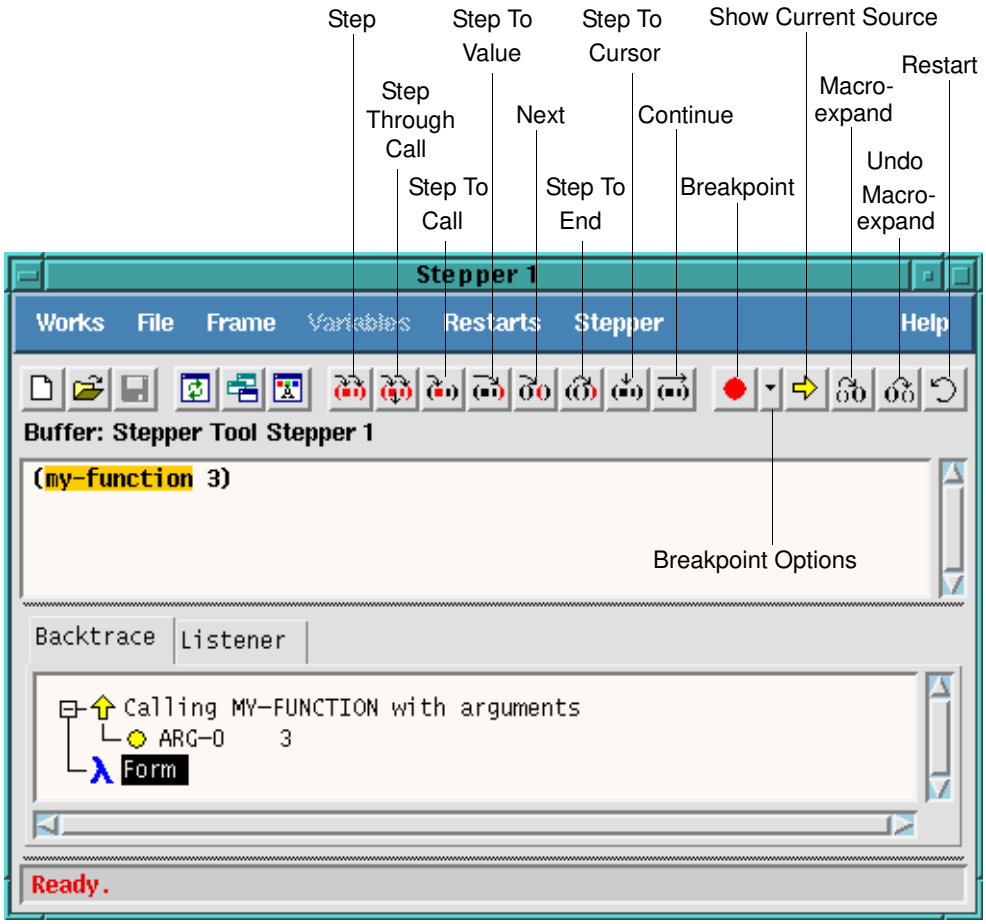
25.4 Stepper controls

The **Stepper** menu offers fine control over the next step.

It also includes commands for setting breakpoints, displaying the source code, macroexpansion, and aborting from the current step.

All these commands are also available on the Stepper toolbar as shown in Figure 25.4.

Figure 25.4 The Stepper controls



The Stepper controls operate as described below. Recall that the current position is always highlighted in orange:

Step — Steps once, remaining inside the current form. At the start of the current form, this steps the first inner form. At a function call, it steps to the value. At the form value, it steps the next form or value.

Step Through Call — Steps once. This is the same as Step above, except that at a function call, it steps that function if the source is known.

Step To Call — At the start of a form, steps to the function call of that form after evaluation of the arguments. At a function call or at the end of a form, steps to the function call of the enclosing form.

Step To Value — At the start or function call of a form, steps to the value of that form. At the end of a form, steps to the value of the enclosing form.

Next — Steps to the start of the next form, or behaves like Step if there is no next form.

Step To End — Steps to the value of the current function.

Step To Cursor — Steps to the cursor position, or displays a message if that position is not steppable.

Continue — Runs the code until a breakpoint is reached.

Breakpoint — Sets a breakpoint at the position of the cursor if there is no breakpoint there already and the position is steppable. If there is a breakpoint under the cursor, this command removes it. Note that breakpoints are highlighted red, though the orange highlight on the current stepping position overrides any breakpoint highlight..

Show Current Source — Moves the editor buffer in the source area so that the definition at the top of the backtrace area, and the active form within it, is visible.

Macroexpand — Macroexpands the form under the cursor.

Undo Macroexpand — Collapses the macroexpansion under the cursor.

Restart — Aborts the execution and returns to the form which you first stepped, allowing you to repeat the execution or edit the form. This command is available only when using the Stepper in standalone mode.

Breakpoint Options — The Breakpoint Options menu allows you to set properties of a breakpoint as described in "Breakpoints" on page 350.

25.4.1 Shortcut keys for the Stepper

The following Editor commands run the corresponding Stepper command in the current stepper:

```
Stepper Breakpoint
Stepper Continue
Stepper Macroexpand
Stepper Next
Stepper Restart
Stepper Show Current Source
Stepper Step
Stepper Step Through Call
Stepper Step To Call
Stepper Step To Cursor
Stepper Step To End
Stepper Step To Value
Stepper Undo Macroexpand
```

These commands can be bound to keys in the LispWorks editor, which makes those keys invoke the command in a Stepper tool. For example:

```
(editor:bind-key "Stepper Step" #("Control-S" Control-s"))
```

Note: the editor key binding only takes effect when the input focus is in the Source or Listener panes of the Stepper tool.

For more information about Editor key bindings, see the *LispWorks Editor User Guide*.

25.5 Stepper restarts

The **Restarts** menu lists a number of restart options, which offer ways to continue execution.

This works the same as described for the Debugger tool in “Simple use of the Debugger tool” on page 123.

25.6 Breakpoints

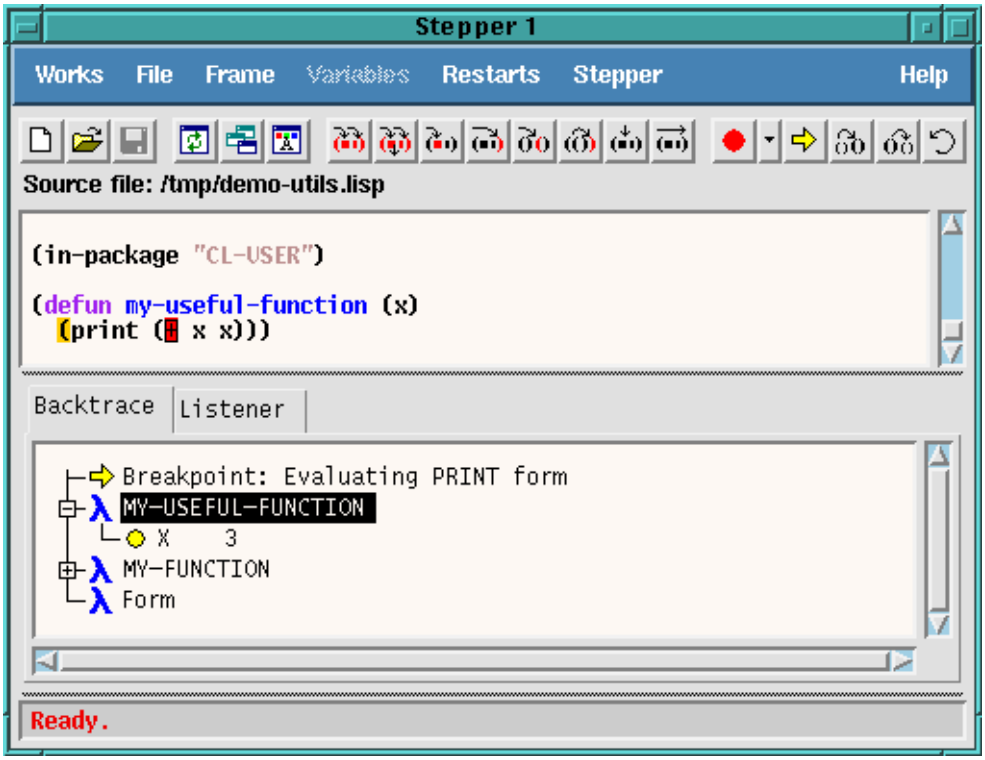
You can set a breakpoint in any form that might be evaluated, except for atomic and constant forms. The breakpoint can be at the start, function call or return point of the form.

When not at the current stepping position, a breakpoint is highlighted red in the Stepper source area. When the same source code is also visible in an Editor tool, the breakpoint is visible there too.

25.6.1 Setting breakpoints

To set a breakpoint from the Stepper, position the cursor where you want the breakpoint and choose **Stepper > Breakpoint** or click in the Stepper toolbar.

Figure 25.5 A breakpoint on the function call +



When you run code, or choose **Stepper > Continue**, execution stops if a breakpoint is reached. The Stepper will show the form in the source area with the breakpoint highlighted in yellow.

In the picture above, execution has stopped at the start of the print form and we have just set a breakpoint on the call to +. Continuing from this point will cause execution to stop just before it calls +, and the Stepper will display the arguments that are about to be passed to +.

If you set a breakpoint on the closing parenthesis of a form, then it will cause execution to stop when the form returns and the top backtrace frame will display the values of that form.

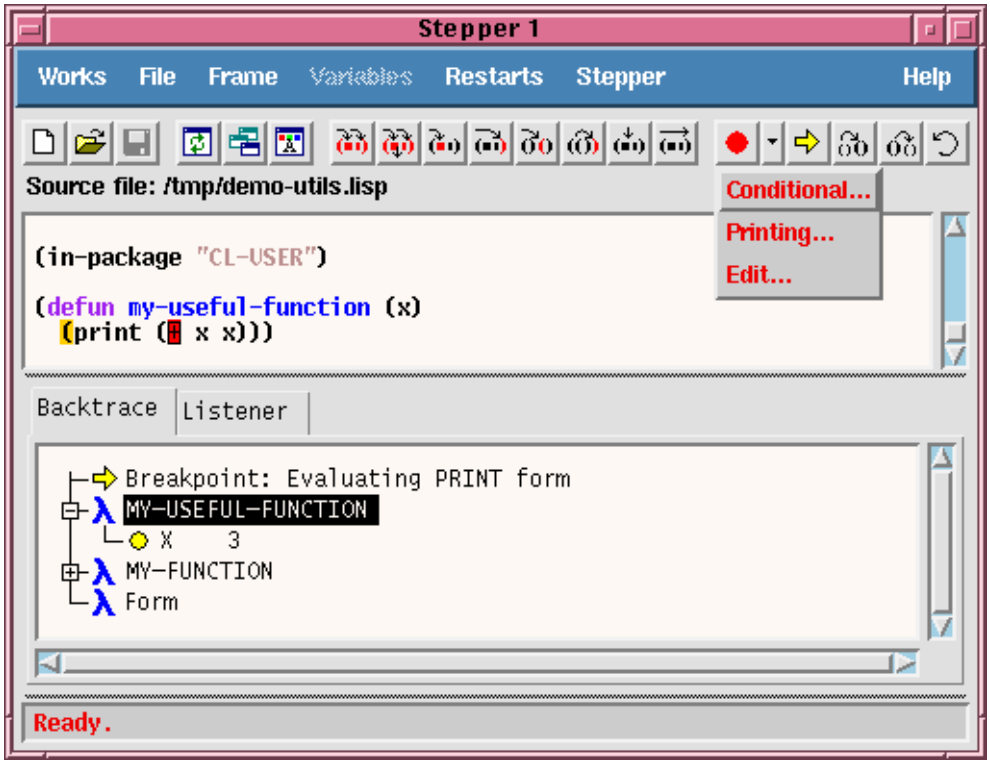
To set a breakpoint from the Editor, see “Breakpoints” on page 181.

25.6.2 Conditional breakpoints

A breakpoint can be modified to make it effective only when a condition is true.

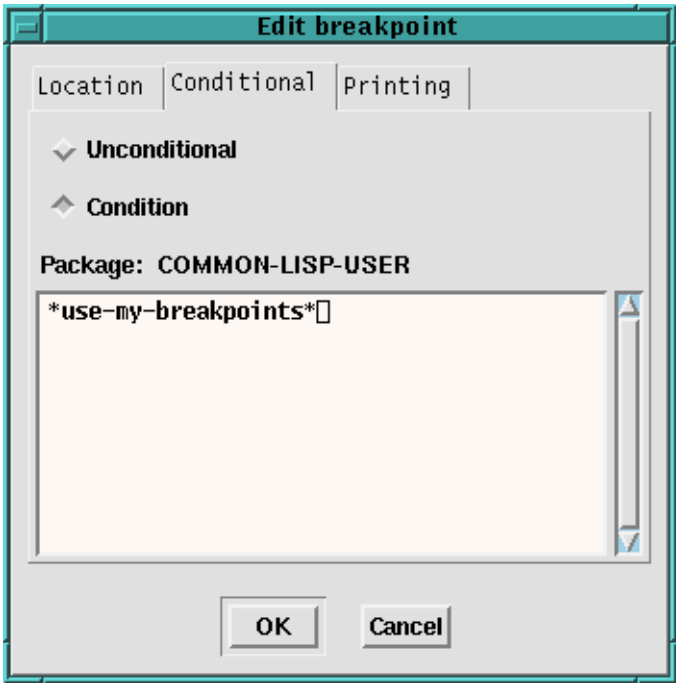
Suppose that you have reached a breakpoint on the call to **+** as set in the example above. To make this breakpoint conditional, choose **Conditional...** from the Breakpoint Options menu:

Figure 25.6 The Breakpoint Options menu



Define your variable **`*use-my-breakpoints*`** and enter it in the Condition area of the Edit Breakpoint dialog, and click **OK**.

Figure 25.7 The Edit Breakpoint dialog



The form defining the breakpoint condition is evaluated in the package where the stepped function was defined. Note that this package is displayed in the Conditional tab of the Edit Breakpoint dialog. Therefore, after confirming the dialog shown above, your code breaks at the breakpoint depending on the value of **`common-lisp-user::*use-my-breakpoints*`**.

To make a breakpoint unconditional, select **Unconditional** in the dialog shown above.

Note: you cannot currently access the values of local variables in the condition expression.

25.6.3 Printing breakpoints

A breakpoint can be modified to make it print an expression and its value when it is reached.

Again suppose that you have reached a breakpoint on the call to **+** as set in the example above. To make this breakpoint print, choose **Printing...** from the Breakpoint Options menu: Enter a valid Lisp expression in the Printing tab of the Edit Breakpoint dialog, and click OK.

When the breakpoint is reached, the expression and its value are printed like this:

```
Stepper value (+ 4 4 4 4): 16
```

The Lisp expression is evaluated in the package where the stepped function was defined. Note that this package is displayed in the Printing tab of the Edit Breakpoint dialog.

If you check the **Print without stopping** option, then the above line is printed but the code continues to execute and does not stop at the breakpoint.

Note: you cannot currently access the values of local variables in the printed expression.

25.6.4 Editing breakpoints

To edit the Conditional or Printing properties of a breakpoint visible in the source, position the cursor on the breakpoint and proceed as described in “Conditional breakpoints” on page 352 or “Printing breakpoints” on page 353.

Where you wish to change the Conditional or Printing properties of a breakpoint without finding it in the source, choose **Edit...** from the Breakpoint Options menu or the menu command **Stepper > Edit Breakpoints...**. Select a breakpoint in the Breakpoints list and click the **Edit...** button. Choose the Conditional or Printing tab as appropriate and proceed as described in “Conditional breakpoints” on page 352 and “Printing breakpoints” on page 353.

To visit the source code where a breakpoint was set, choose **Edit...** from the Breakpoint Options menu or the menu command **Stepper > Edit Breakpoints...**. Select a breakpoint in the Breakpoints list and press the **Goto Source** button. This cancels the dialog and then displays the source containing the breakpoint.

25.6.5 Removing breakpoints

To remove a breakpoint under the cursor, click in the toolbar. Equivalently you can choose **Stepper > Breakpoint**.

Where you wish to remove one or more breakpoints without finding them in the source, choose **Edit...** from the Breakpoint Options menu or the menu command **Stepper > Edit Breakpoints...**, select a breakpoint or breakpoints in the Breakpoints list, and click **Remove**.

25.7 Stepping macro forms

Where your code contains a macro, you can step the macroexpansion or simply step the macro form as-is.

25.7.1 Interactive macroexpansion

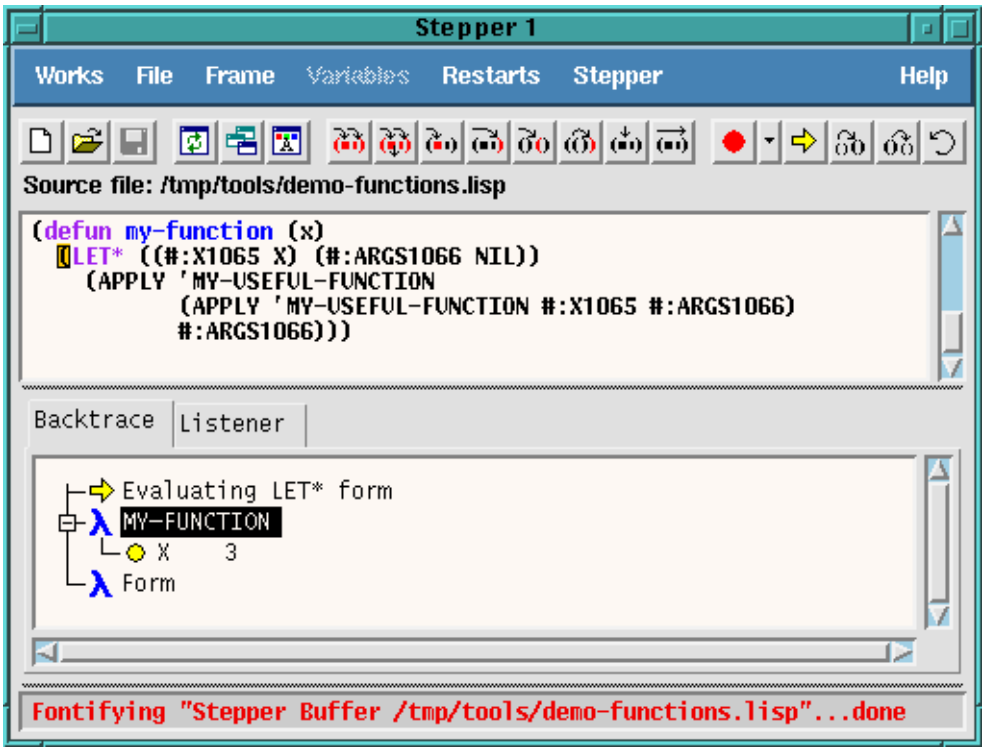
When the Stepper reaches code for which the source contains an unexpanded macro form, by default it offers you the option of macroexpanding that form.

To see this, follow the example in “Standalone use of the stepper” on page 342 and when your reach `my-function` choose **Stepper > Step Through Call** or click in the Stepper toolbar.

The source code for `my-function` is shown in the Source area of the Stepper. Choose **Stepper > Step** or click in the Stepper toolbar.

Click **Yes** on the dialog asking "Expand MY-MACRO form?". The macroexpansion replaces the macro form:

Figure 25.8 Stepping a macroexpansion



Now you can Step into the macroexpansion of **`my-macro`**.

25.7.2 Macroexpansion in the stepper

To macroexpand a macro form before reaching it in the Stepper, position the cursor at the start of the macro form and choose the menu command **Stepper > Macroexpand** or click in the Stepper toolbar. You can only this when the Stepper has already stepped the function.

Sometimes it is useful to expand macros in outer forms, to allow the more detailed stepping of their expansions. For example, for a definition such as

```
(defstruct foo (x (print 10)) y)
```

when stepping

```
(make-foo)
```

expanding the **defstruct** form allows you to step more of the constructor.

25.7.3 Collapsing macroexpansions

To collapse a macroexpansion in the Stepper, position the cursor at the start of the macroexpansion and choose the menu command **Stepper > Undo Macroexpand** or click in the Stepper toolbar.

25.7.4 Controlling macroexpansion

You can alter the way the Stepper handles macro forms on a per-symbol or per-package basis. For instance, you can specify that the Stepper always expands your macros automatically, without prompting. For details, see “Operator preferences” on page 359.

25.8 Listener area

Select the Listener tab of the Stepper tool to display a Listener.

This area offers all the usual Listener and Debugger commands. Moreover, the execution environment is that of the function currently being stepped, and contains the variables of each frame on the stack.

The Stepper listener also offers the following listener commands to control stepping.

:s, **:step** Step Steps once, remaining inside the current form. At the start of the current form, this steps the first inner form. At a function call, it steps to the value. At the form value, it steps the next form or value.

:st, **:step-through-call**

Step Through Call

:sc, **:step-to-call**

Step To Call At the start of a form, steps to the function call of that form after evaluation of the arguments. At a function call or at the end of a form, steps to the function call of the enclosing form.

:sv, **:step-to-value**

Step To Value At the start or function call of a form, steps to the value of that form. At the end of a form, steps to the value of the enclosing form.

:sn, **:next** Next Steps to the start of the next form, or behaves like Step if there is no next form.

:se, **:step-to-end**

Step To End Steps to the value of the current function.

:c, **:continue** Continue Runs the code until a breakpoint is reached.

:sm, **:macroexpand**

Macroexpand Macroexpands the form under the cursor.

:restart Restart Aborts the execution and returns to the form which you first stepped, allowing you to repeat the execution or edit the form. This command is available only when using the Stepper in standalone mode.

See “Stepper controls” on page 346 for a full description of these controls.

25.9 Configuring the Stepper

To configure the Stepper tool, raise the Stepper Preferences dialog, by choosing **Works > Tools > Preferences...** or clicking in the Stepper toolbar.

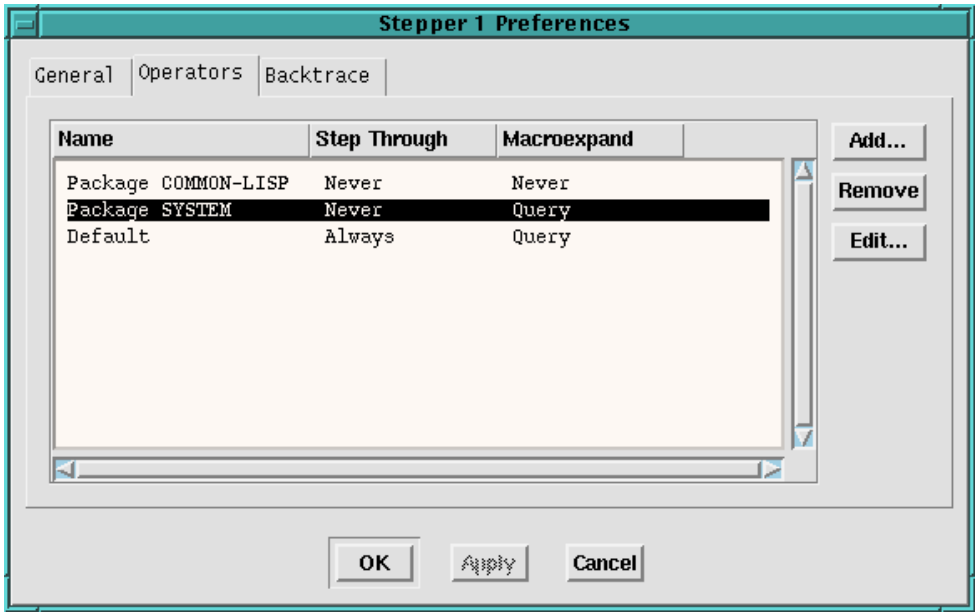
The Stepper Preferences dialog has three tabs:

- The General tab controls display of the Stepper toolbar, as described in “Toolbar configurations” on page 24.

- The Operators tab contains options controlling the behavior when the stepper sees functions or macros in the source.
- The Backtrace tab controls the amount of information shown automatically in the Backtrace area.

25.9.1 Operator preferences

Figure 25.9 Stepper Preferences dialog



When reaching a function call you can use the Step Through Call command to step through the call into its definition. You can configure the Stepper to do this automatically, never do this or ask you which action to take.

Similarly when reaching a macro form you can macroexpand it (or not). You can configure the Stepper to macroexpand automatically, never macroexpand or ask you whether to macroexpand..

For a given symbol naming a function or macro, the action is determined by the preferences in the Operators tab. If the symbol is listed, then the corresponding action is taken. Otherwise, if the symbol's package is listed, then the corresponding action is

taken. If neither the symbol nor its package are shown,. then the default action is taken.

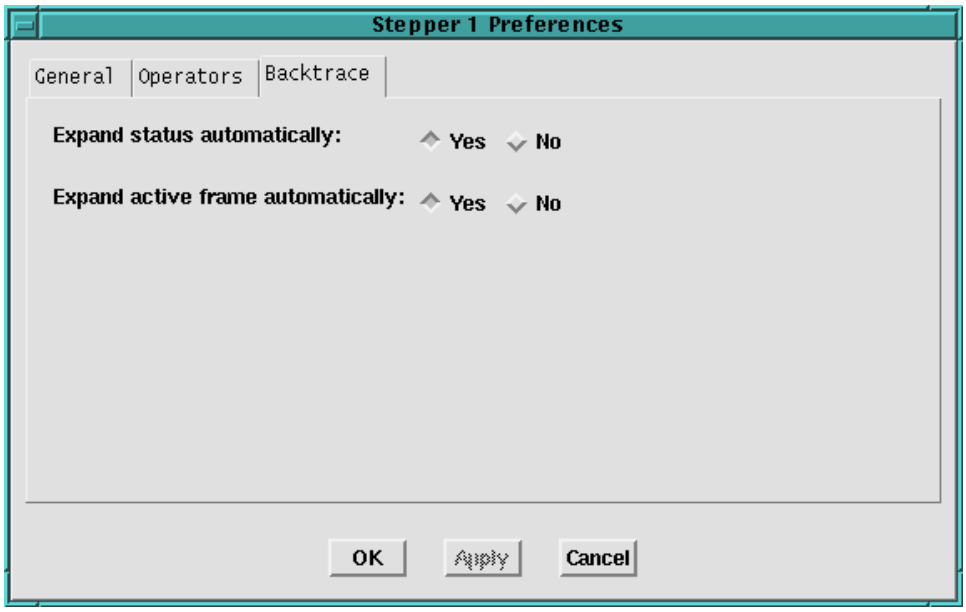
For example, the default behavior on reaching your macro forms is to prompt for whether to macroexpand. To configure the Stepper such that macros defined in the **CL-USER** package are macroexpanded automatically, click the **Add...** button, enter **CL-USER** in the Name pane of the dialog, select **Always** in the **Expand macros** panel, click **OK** and click **OK** to dismiss the Stepper Preferences dialog.

To configure the Stepper such that it never steps through **my-function**, raise the Stepper preferences dialog again, click the **Add...** button and select the **Symbol** radio button. Enter **cl-user::my-function** in the Name pane of the dialog, select **Never** in the **Step through calls** panel, click **OK** and click **OK** to dismiss the Stepper Preferences dialog.

25.9.2 Backtrace preferences

To control the amount of information displayed automatically in the Backtrace area, select the Backtrace tab of the Stepper Preferences dialog:

Figure 25.10 Stepper Preferences Backtrace tab



By default the status item in the Backtrace area automatically expands to show the arguments or return values. To change this behavior, select **No** against **Expand status automatically**.

By default the active frame in the Backtrace area automatically expands to show the local variables and arguments. To change this behavior, select **No** against **Expand active frame automatically**.

Compatibility Note: in LispWorks 5.0 these Backtrace options have the opposite default values. This is changed in LispWorks 5.1 and later versions.

26

The System Browser

26.1 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. See the *LispWorks User Guide* for more on the use of **defsystem**. Once defined, operations such as loading, compiling and printing can be performed on the system as a whole. The system tool ensures that these operations are carried out completely and consistently, without doing unnecessary work, by providing you with a GUI front end for **defsystem**.

A system may itself have other systems as members, allowing a program to consist of a hierarchy of systems. Each system can have compilation and load interdependencies with other systems, 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.

26.2 A brief introduction to systems

A system is defined with a **defsystem** form in an ordinary Lisp source file. This form must be evaluated in the Lisp image in order to use the system.

Once defined, operations can be carried out on the system by invoking Lisp functions.

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 defined 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.

26.2.1 Examples

Consider an example system, **demo**, defined as follows:

```
(defsystem demo (:package "USER")
  :members ("macros"
            "demo-utils"
            "demo-functions")
  :rules ((:in-order-to :compile ("demo-utils" "demo-functions")
            (:caused-by (:compile "macros"))
            (:requires (:load "macros")))))
```

This system compiles and loads members in the **USER** package if the members themselves do not specify packages. The system contains three members—**macros**, **demo-utils**, and **demo-functions**—which may themselves be either files or other systems. There is only one explicit rule in the example. If **macros** needs to be compiled (for instance, if it has been changed), then this causes **demo-utils** and **demo-functions** to be compiled as well, irrespective of whether they have themselves changed. In order for them to be compiled, **macros** 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 **macros**, then *every* member gets compiled. If the changed member is not **macros**, then **macros** 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.lisp** 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**.

26.3 The System Browser

The System Browser provides an intuitive graphical way to examine and operate on systems and their members.

For example, the operation outlined in "A brief introduction to systems" on page 364 would be performed by the System Browser menu commands **Systems > Compilation options > Force** followed by **Systems > Compile**.

To create a System Browser, choose **Works > Tools > System Browser** or press in the Podium. Alternatively, choose **File > Browse Parent System** from any appropriate tool in the environment or execute **Meta+X Describe System** in an editor, to display the parent system for the selected or current file in the System Browser. See "Operating on files" on page 31 for details.

In order to browse a system, first ensure it is defined. To define a system, load the Lisp source code containing the **defsystem** form into the Lisp image. For instance, open the file in an Editor and choose **File > Load**. Alternatively, choose **File > Load...** from the System Browser and choose a file to load in the dialog that appears.

26.4 A description of the System Browser

The System Browser has four views:

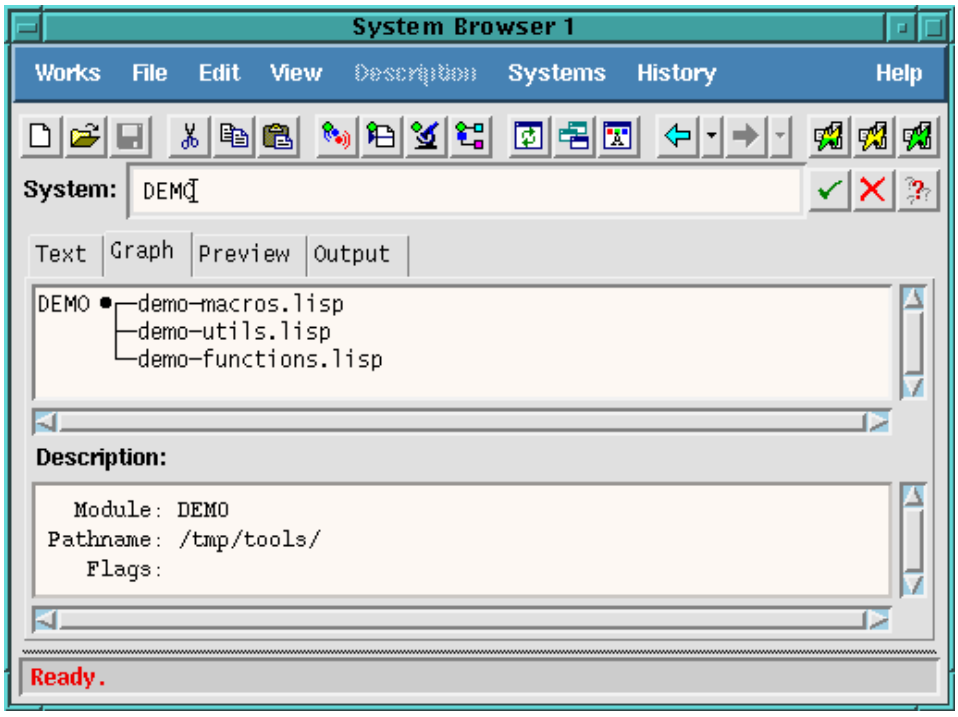
- The graph view displays a graph of all the systems defined in the image, together with their members.
- The text view lists the systems defined in the image together with the members of the current system.

- The preview view provides a powerful way of generating and executing systems plans.
- The output view is used to display any output messages which have been created by the System Browser as a result of executing plans.

26.5 Examining the system graph

When you first invoke the System Browser, the graph view is the default view. You can also switch to it from another view by choosing the relevant tab above the main view. The graph view is shown in Figure 26.1 below.

Figure 26.1 Displaying loaded systems using the graph view



The System Browser window has four areas, described below.

26.5.1 System area

The *System area* is used to enter and display in the name of the system.

You can browse a system by entering its name into the **System:** area. Whilst doing this you can press **Up**, **Down** or click to complete a partially specified name. This allows you to select from a list of all system names which begin with the partial input you have entered. See “Completion” on page 46 for detailed instructions.

The members of the system are displayed in the graph area.

26.5.2 Graph area

The *Graph area* produces a graph of the current system, together with all its membersThe generic facilities available to all graph views throughout the environment are available here; see Chapter 5, “Manipulating Graphs” for details.

- Double-click on a filename to display the file in the editor.
- Click on an unfilled circle alongside a system name to display its members.
- Click on a filled circle alongside a system name to hide its members.
- Select either a system name or a file name to display details in the Description area.

You can operate on systems and files via the context menu, which offers commands such as **Concatenate...** and **Search Files...** for systems, and **Compile** and **Print...** for files. The system commands are also available in the **Systems** menu. If no items are selected, the commands apply to the current system, whose name is printed in the System area.

The standard graph layout menu, described in “Manipulating Graphs” on page 57, is available as the **Graph** submenu of the context menu.

To traverse the system hierarchy, double click a system node in the graph. If the destired parent node is not in the graph, choose **Systems > Parent**. The parent of all systems defined in the image at any time is called the **ROOT-SYSTEM**.

26.5.3 Description area

The *Description area* shows details about any system member selected in the Graph area. The following items of information are shown:

Module	The name of the selected member. This is either the filename (if the member is a file of source code) or the system name (if the member is a subsystem).
Pathname	The directory pathname of the selected member. This is the full pathname of the file, if the selected member is a file of source code, or the default directory of the system, if the selected member is a subsystem.
Flags	This lists any keyword flags which have been set for the selected member in the system definition, such as the `:source-only` flag.

To operate on any of the items displayed in this area, select them and choose a command from the **Description** menu, which contains the standard actions described in "Performing operations on selected objects" on page 35. By making multiple selections, you can operate on as many of the items as you like.

26.5.4 Performing operations on system members

A variety of operations can be performed on any number of nodes selected in the Graph area. If no system nodes are selected, or if you are in another view, the commands are performed on the current system, whose name is printed in the System area.

The **Systems** menu gives you access to the standard actions described in "Performing operations on selected objects" on page 35.

- Choose **Systems > Parent** to browse the parent system. This takes you up one level in the hierarchy.
- Choose **Systems > Compile and Load**, **Systems > Compile**, or **Systems > Load** to compile or load the selected systems.
- Choose **Systems > Concatenate...** to produce a single fasl file from a system. You will need to supply the name of the fasl file, when prompted.

- Choose **Systems > Search Files...** to search the files of the selected systems (and any subsystems) for a given regular expression. A dialog prompts for the regular expression, and then a Search Files tool is raised in System Search mode, displaying the results of the search. The Search Files tool is described in “The Search Files tool” on page 207.
- Choose **Systems > Hide Files**, to remove system member files from the graph and display only systems. Choose **Systems > Show Files** to reverse this effect.
- Choose **Systems > Replace** to search all the files in the selected members (and any subsystems) for a given string and replace it with another string. You are prompted for both strings in the echo area.

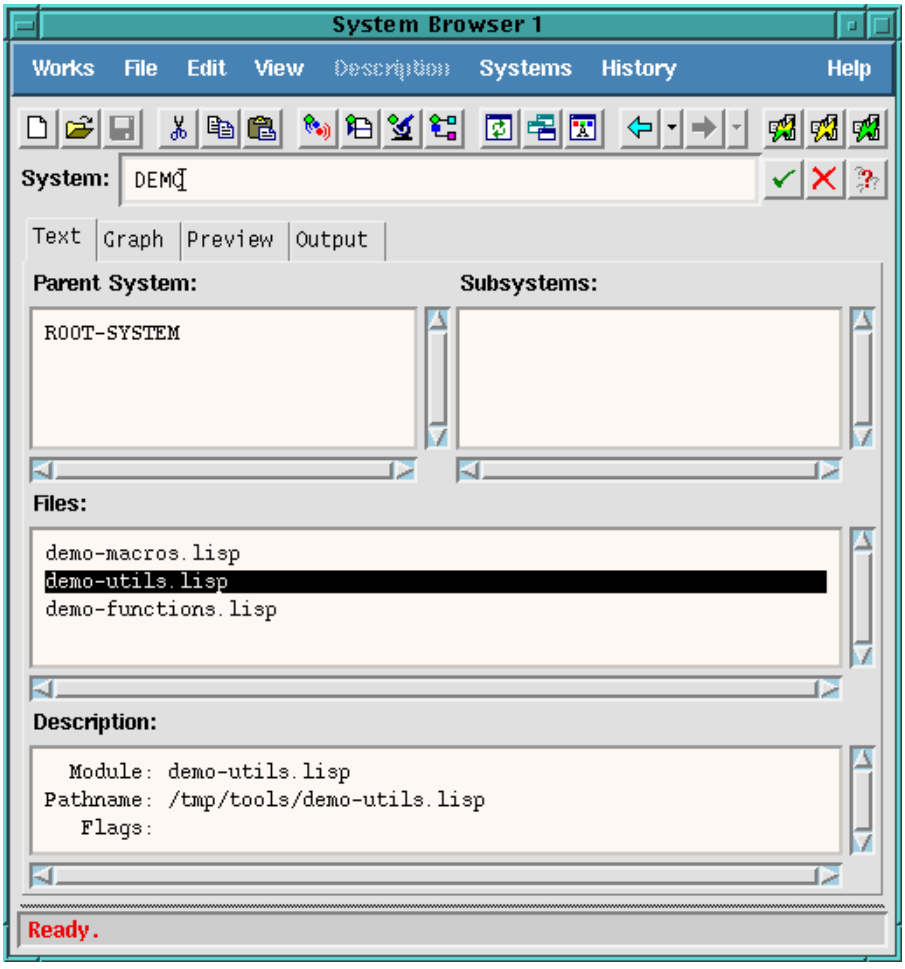
  You need to save the buffers to actually save the changes on disk, this is easily done using the Editor tool - see “Buffers area” on page 149 for details.

26.6 Examining systems in the text view

The text view allows you to list the parent system, subsystems and files in the current system in one view, and gives you an easy way of changing the current system.

Choose the Text tab to display this view. The System Browser appears as shown in Figure 26.2 below.

Figure 26.2 Displaying loaded systems using the text view



The System Browser contains the areas described below when in the text view.

26.6.1 System area

As with the graph view, the current system is shown here. See “System area” on page 368 for details about this area.

26.6.2 Parent system area

This area lists any parent systems of the current system. Note that every system apart from the **ROOT-SYSTEM** must have at least one parent.

Double-click on any item in this list to make it the current system. Its name is printed in the System area.

26.6.3 Subsystems area

This area lists any systems which are subsystems of the current system.

Double-click on any item in this list to make it the current system. Its name is shown in the System area.

26.6.4 Files area

This area lists any files which are members of the current system. Source files containing either Lisp or non-Lisp code (such as C code which is loaded via the Foreign Language Interface) are listed in this area.

- Select a file to display its description in the Description area.
- Double-click on a file to display it in the editor.

26.6.5 File description area

The File Description area displays information about any system member selected in the Files area. If no such member is selected, information about the current system (the one named in the System area) is shown instead. The same pieces of information are shown as in the graph view. See “Description area” on page 369 for details. As with other views, items selected in this area can be operated on using commands in the **Description** menu.

26.7 Generating and executing plans in the preview view

The preview view allows you to generate different system plans automatically based on three things:

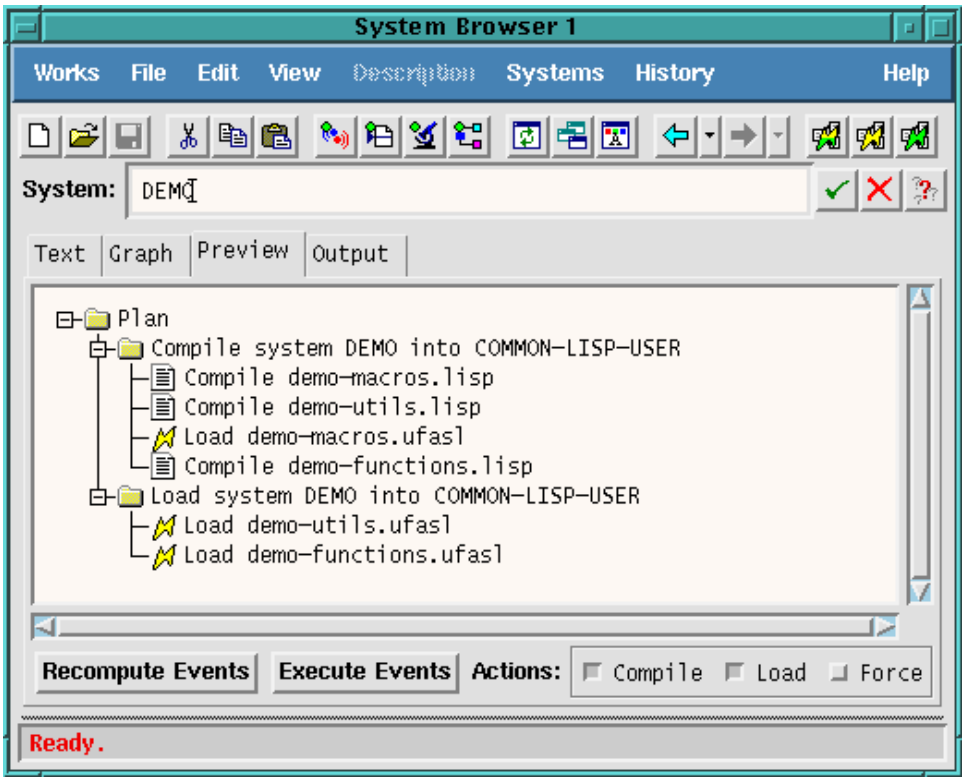
- The current compilation and load status of each member of a system.

- The rules specified in the system definition.
- The specific actions that you wish to perform.

You can use this view to browse the plan and to execute all or any part of it, as well as generate it.

Click on the Preview tab to switch to the preview view in the System Browser. The System Browser appears.

Figure 26.3 Previewing system plans using the preview view



Click **Recompute Events** or the menu command **Works > Refresh** and expand nodes in the tree to make the plan fully visible as in Figure 26.3.

The System Browser has the areas described below.

26.7.1 System area

As with the graph view, the current system is shown here. See “System area” on page 368 for details about this area.

26.7.2 Actions area

The *Actions area* contains a number of options allowing you to choose which actions you want to perform, thereby allowing you to create system plans.

The **Compile**, **Load** and **Force** check buttons can be selected or deselected as desired. Note that at least one of **Compile** and **Load** must always be selected.

- Select **Compile** to create a plan for system compilation. The plan displays what actions need to be performed in order to update the fasls for the entire system.
- Select **Load** to create a plan for loading the system. The plan displays a list of the actions required to load the system.
- Select **Force** if you want to force compilation or loading of all system members, whether it is necessary or not.

Click **Recompute Events** to create a new plan for the specified options. You should click this button whenever you change the **Compile**, **Load**, or **Force** options, or whenever you change any of the files in the system or any of its subsystems.

Click **Execute Events** is used to execute the events currently selected in the main area. Notice that this button is only enabled.when some event is selected in the plan. See “Executing plans in the preview view” below for details.

26.7.3 Filter area

As with other tools, you can use the Filter area to restrict the output in the plan area to just those actions you are interested in. This may be useful, for instance, if you want to see only compile actions, or only load actions, or if you are only interested in the actions that need to be performed for a particular file.

26.7.4 Plan area

The Plan area lists the actions in the current plan. Items are indented to indicate groups of related actions. Thus, if a subsystem needs to be loaded, the individual files or subsystems that comprise it are listed underneath, and are indented with respect to it.

26.7.5 File description area

The File Description area displays information about any system member selected in the Plan area. If no such member is selected, information about the current system (the one named in the System area) is shown instead. The same pieces of information are shown as in the graph view. See Section 26.5.3 on page 369 for details. As with other views, items selected in this area can be operated on using commands in the **Description** menu.

26.7.6 Executing plans in the preview view

Once you have created a plan in the preview view, there are a number of ways that you can execute either the whole plan, or individual actions within that plan.

As already mentioned, to execute individual actions in the plan, select them in the main area and then click the **Execute Events** button.

To execute the whole plan, just choose the relevant command:

- Choose the menu command **Systems > Load** or click the button to execute a plan for loading the system.
- Choose the menu command **Systems > Compile** or click the button to execute a plan for compiling the system.
- Choose the menu command **Systems > Compile and Load** or click the button to execute a plan for both compiling and loading the system.

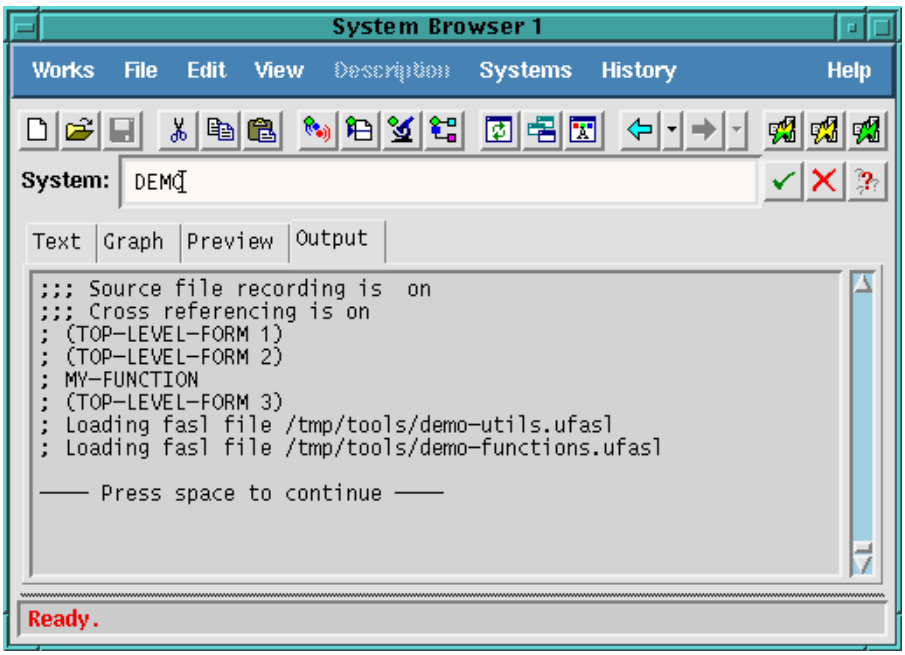
Note that you can also execute the whole plan by choosing **Edit > Select All** and then clicking **Execute Events**.

26.8 Examining output in the output view

The output view can be used to view any messages that have been generated as a result of actions performed in the System Browser. This largely consists of compilation and load messages that are generated when system plans or individual actions in a plan are executed.

Click on the Output tab to switch to the output view. The System Browser appears as in Figure 26.4.

Figure 26.4 Viewing output in the System Browser



The output view has the areas described below.

26.8.1 System area

As with the graph view, the current system is shown here. See “System area” on page 368 for details about this area.

26.8.2 Output area

The largest area in this view is used to display all the output messages which have been generated by the System Browser. This area has the same properties as the Output Browser described in Chapter 21, “The Output Browser”. In particular you can interact with highlighted compiler warnings and notes in the same way as in any output tab in the IDE.

26.9 Configuring the display

The System Browser allows you to configure the display so that it best suits your needs. The commands available for this are described below.

26.9.1 Sorting entries

Entries in the System Browser can be sorted in a number of ways. Choose **Works > Tools > Preferences...** or click to display the System Browser Preferences dialog. Click on the General tab to view the sorting options.

By Name	Sorts entries in the main area of the current view (the graph in the graph view and the Files area in the text view) according to the symbol name.
By Package	Sorts entries in the main area according to their package.
Unsorted	Leave entries in the main area unsorted. This is the default setting.

26.9.2 Displaying package information

As with other tools, you can configure the way package names are displayed in the System Browser, using the Package box. See “Displaying packages” on page 32 for full details.

26.9.3 Display of the toolbar

You can control whether the System Browser displays the compile/load and history toolbars by the option **Show Toolbar**, as described in “Toolbar configurations” on page 24.

26.9.4 Graph preferences

You can configure the graph in the Graph tab of the System Browser similarly to other tools. For details see “Preferences for graphs” on page 63.

26.10 Setting options in the system browser

The **Systems > Compilation Options** menu allows you to set options which apply whenever you compile or load system members. Each of the commands described below toggles the respective option.

Choose **Systems > Compilation Options > Force** to force the compile or load operation to be performed. If you are operating on a whole system (as opposed to system members which are files) this means that actions for all the members are added to the plan.

Choose **Systems > Compilation Options > Source** to force the use of Lisp source rather than fasls in operations on the system.

Choose **Systems > Compilation Options > Preview** to automatically preview the plan prior to execution of a compile or load instruction chosen from the **Systems** menu. This switches the System Browser to the preview view and allows you to see what operations are going to be performed, and to change them if you want. See “Generating and executing plans in the preview view” on page 372 for full details about previewing plans.

Choose **Systems > Concatenate...** to concatenate the selected system into a single fasl after compiling it. You will need to supply the name of the single fasl file, when prompted.

27

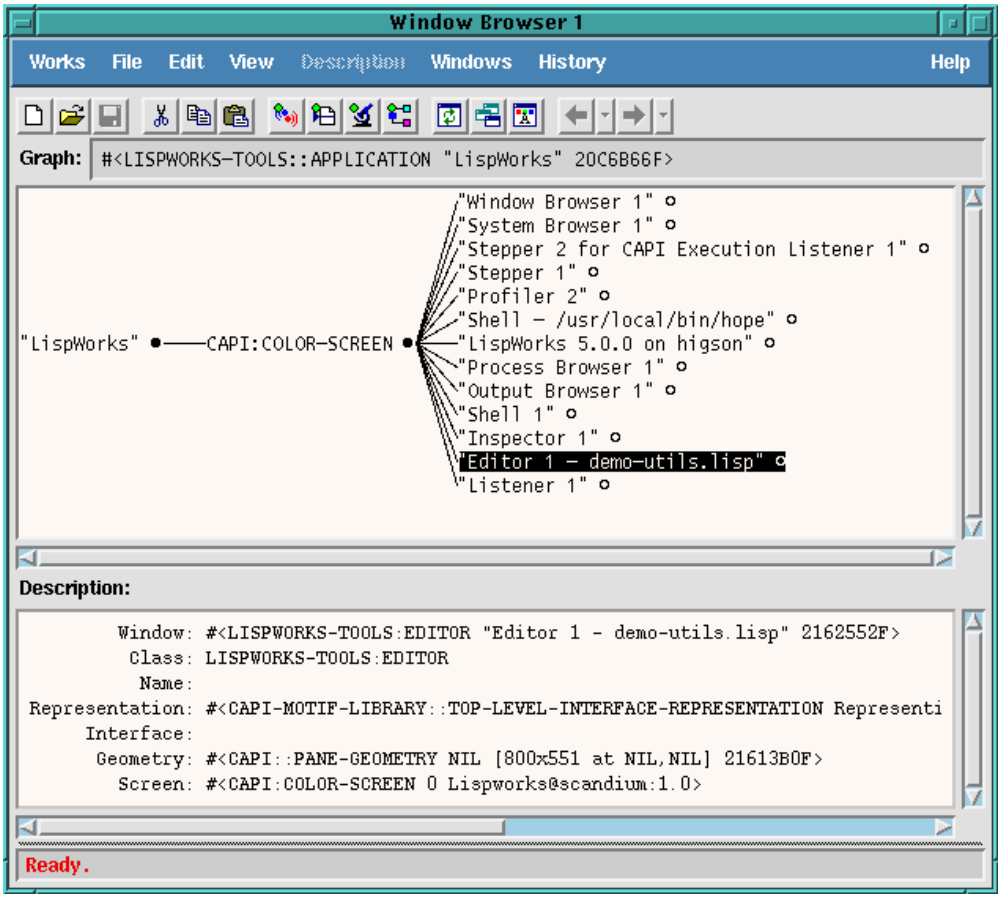
The Window Browser

27.1 Introduction

The Window Browser lets you examine any windows that have been created in the environment. You can examine not only the environment windows themselves, but also more discrete components of those windows, such menus and menu commands. To create a Window Browser, choose **Works > Tools > Window Browser** or click in the Podium.

The Window Browser only has one view, shown in Figure 27.1.

Figure 27.1 The Window Browser



The Window Browser has three sections.

27.1.1 Graph box

This text box shows the window object that is being examined; that is, the name of the window at the root of the graph.

27.1.2 Window graph

The window graph displays the current window and all its subwindows. The generic facilities available to all graphs throughout Common LispWorks are available here; see Chapter 5, “Manipulating Graphs” for details.

When you first create a Window Browser, it automatically browses the parent window of the whole environment. A graph of the parent window together with its children—each individual window that has been created—is drawn in the main area.

Select any item in the graph to display its description in the Description area.

To see the children of an unexpanded node in the graph, click on the unfilled circle to its right. To make one of the child windows be the root of the graph, select it and choose **Windows > Browse - Window**.

Any items selected in the graph can be operated on using commands in the **Windows** menu. If no items are selected, the commands in this menu apply to the root window of the graph. See Section 27.3 on page 385 for details.

27.1.3 Description list

The Description list gives a description of the item selected in the Graph area. If nothing is selected, a description of the window at the root of the graph is shown. The following information is listed:

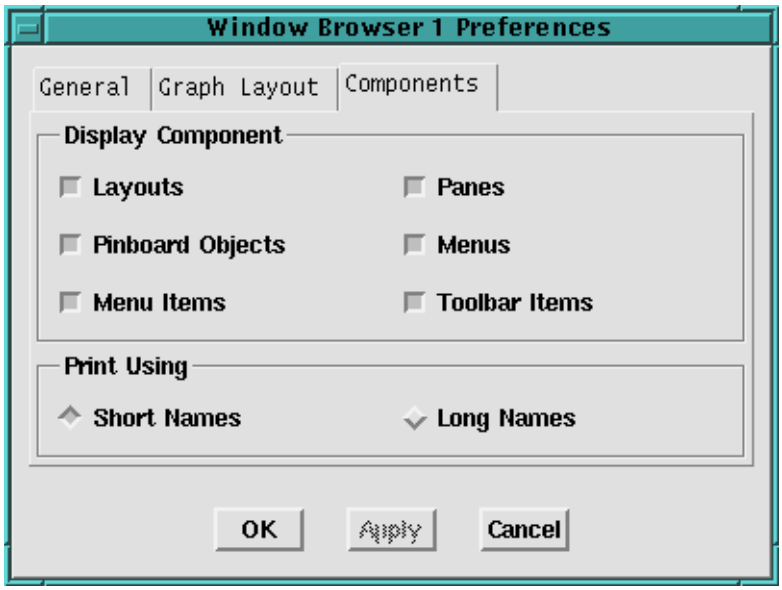
Window	The object which represents the selected window
Class	The class of the window object.
Name	The name of the selected window.
Representation	The CAPI representation of the selected window.
Interface	The underlying native window system object which represents the selected window.
Screen	The name of the screen on which the selected window is displayed.

Any item selected in the Description list can be operated on by using commands under the **Description** menu. This menu gives you access to the standard actions commands described in Section 3.7 on page 35.

27.2 Configuring the Window Browser

Using the Window Browser Preferences dialog, shown in Figure 27.2, you can configure the Window Browser so that it best suits your needs. Choose **Works > Tools > Preferences...**. or click to display this dialog.

Figure 27.2 Window Browser Preferences dialog



The Window Browser Preferences dialog has three tabs:

- The General tab contains options for configuring general properties of the Window Browser.
- The Graph Layout tab contains options for configuring options specific to the graph. See Section 5.6 on page 63 for a description of these options.
- The Components tab contains options for configuring properties unique to the Window Browser.

27.2.1 Sorting entries

Entries in the Window Browser can be sorted using the Sort panel in the General tab of the Preferences dialog. Choose the sort option you require from the list available.

By Name	Sorts items alphabetically by name.
By Package	Sorts items alphabetically by package name.
Unsorted	Displays items in the order they are defined in. This is the default setting.

27.2.2 Displaying package information

As with other tools, you can configure the way package names are displayed in the Window Browser using options available in the General tab.

Check or un-check **Show Package Names** to turn the display of package names in the Window Browser on and off.

Specify the process package of the Window Browser in the text box.

27.2.3 Displaying the toolbar

You can control whether the Window Browser displays its history toolbar by the option **Show Toolbar** in the General tab of the Preferences dialog, as described in “Toolbar configurations” on page 24.

27.2.4 Displaying different types of window

There are several types of window object which can be displayed in the Window Browser, and you can configure which types are displayed using the Display Component panel of the Components tab in the Preferences dialog. Six options are available; select whichever ones you want to display.

Below, the *current window* means the window that is at the root of the graph.

Layouts	Displays the major layouts available to the current window. For the parent window of the environment, this means all the windows that have been created. For an individual window, this means the configuration of the different panes in that window.
Panes	Displays CAPI panes in the current window.
Pinboard Objects	Displays any pinboard objects in the current window. See the *CAPI User Guide* for a full description of pinboard objects.
Menus	Displays any menus available to the current window.
Menu Items	Displays any menu items available to the current window. This option only takes effect if Menus is selected as well.
Toolbar Items	Displays any toolbar items available to the current window.

By default, all these options are selected in the Window Browser.

27.2.5 Displaying short or long names

By default, the Window Browser gives each item in the graph a short name. You can also display the complete symbol name for each item if you wish, as displayed in the Window line of the Description list. You can configure this option from the Components tab of Preferences dialog.

Click **Long Names** in the Print Using panel to display the complete symbol name of each item in the graph.

Click **Short Names** in the Print Using panel to display the short name for each item in the graph. This is the default setting.

Bear in mind that graphs are a lot larger when you display them using long names, and can therefore be more difficult to examine.

27.3 Performing operations on windows

You can perform a number of operations on any windows selected in the Graph area using the commands in the **Windows** menu. If no items are selected in the Graph area, the commands in this menu apply to the root window of the graph.

The **Windows** menu gives you access to the standard actions commands described in Section 3.7 on page 35.

27.3.1 Navigating the window hierarchy

Choose **Windows > Browse Parent** to display the parent of the current window. This takes you back up one level in the window hierarchy.

Choose **Windows > Browse Representation** to examine the representation of the current window. This is discussed in Section 3.14 on page 49.

Choose **Windows > Browse Screens** to examine the parent window of the environment once again—this takes you back up to the root of the window hierarchy.

27.3.2 Window control

There are several commands which give you control over the current window.

Choose **Windows > Lower** to push the current window to the bottom of the pile of windows on-screen.

Choose **Windows > Raise** to bring the current window to the front of your screen.

Choose **Windows > Quit** to quit the any windows selected in the graph.

Choose **Windows > Destroy** to destroy any windows which are selected in the graph. You are prompted before the windows are destroyed.

28

The Application Builder

28.1 Introduction

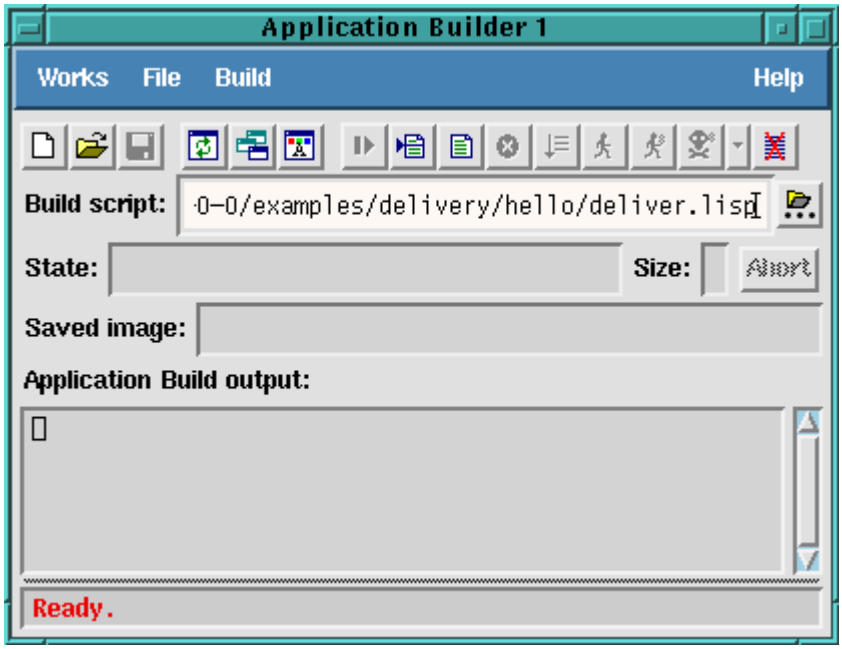
The Application Builder makes it easier to create applications, typically by calling **`deliver`**. This tool helps you to control and debug the delivery process. On Intel Macintosh computers, it also eases the building of universal binaries.

To create an Application Builder, choose **Works > Tools > Application Builder** or click in the Podium.

Note: the Application Builder needs **`deliver`** (or **`save-image`**) functionality and therefore it is not available in LispWorks Personal Edition.

On first use the Application Builder appears all set to build the CAPI example Hello World, as shown in Figure 28.1 below.

Figure 28.1 The Application Builder with the Hello World example



Choose **Build > Build** or click to build the Hello World example.

Then choose **Build > Run** or click to run the Hello World example that you just built.

Note that the Application Builder toolbar commands are also available on the **Build** menu.

28.1.1 What the Application Builder does

This tool helps to control and debug the delivery process.

To use the Application Builder, you need to configure it to know about it your delivery script, and then invoke the Build command. This runs LispWorks in a subprocess with the script. The Application Builder displays the output, and reports on the progress of Delivery. It also allows you to edit the script, and to run the built application.

Note: The Application Builder does not help you in writing your application.

Note: In LispWorks 4.4 and previous versions, you would generally need to write a shell script which runs LispWorks with the appropriate command arguments for delivery. The Application Builder obviates the need for such a script, allowing you to complete the delivery process entirely within the LispWorks IDE.

28.2 Preparing to build your application

First you will need a script which loads your application code and then calls **`deliver`**. Delivery scripts are described in detail in the *LispWorks Delivery User Guide*. If you do not already have a delivery script, the Application Builder can help you to create a simple script, which you can modify as needed.

It is also possible to use the Application Builder with a script that calls **`save-image`** rather than **`deliver`**.

28.2.1 The script

The delivery script is a Lisp source file, which at a minimum loads patches and your application code, and then calls **`deliver`**. The script may do other things, such as configuring your application, though in general you should try to keep it as simple as possible.

28.2.1.1 Using your existing delivery script

If you already have an appropriate delivery script (because you already delivered your application before), click the button to the right of the Build script pane and select your script file. The Application Builder now displays the path to your script in its Build script pane.

28.2.1.2 Creating a new delivery script

Suppose that you already have a file **`compile-and-load-my-app.lisp`** that you use to compile and load your application. Then you can create a suitable delivery script with the help of the Application Builder.

To create the new delivery script:

1. Choose **Build > Make a New Script** or click in the Application Builder toolbar.

   This displays a dialog as shown in Figure 28.2, page 390.

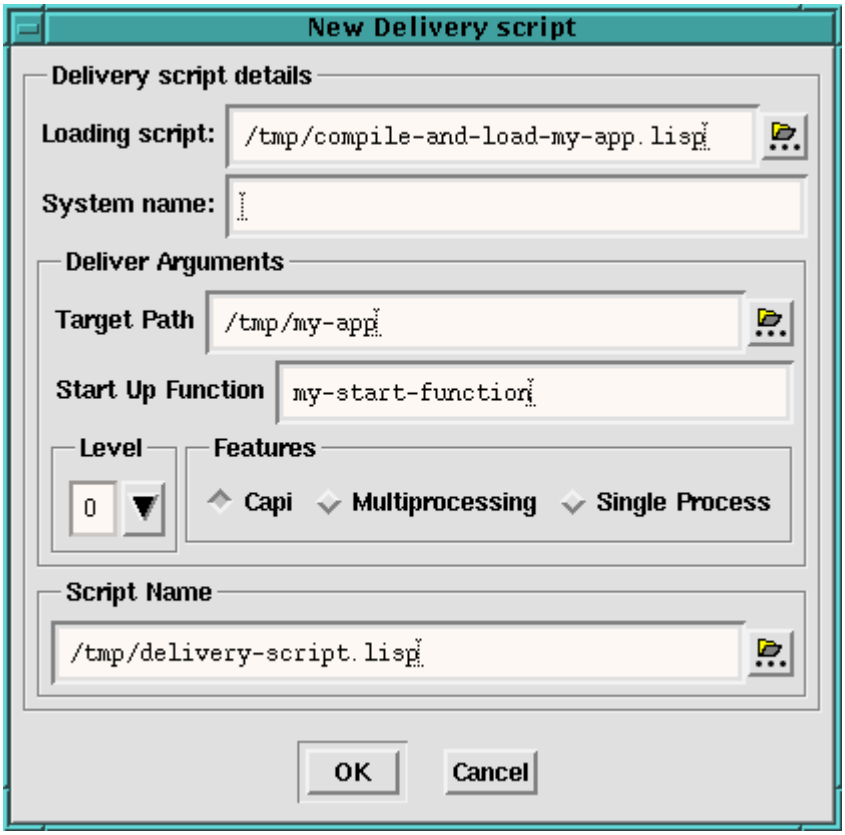
2. Enter the path to `compile-and-load-my-app.lisp` in the Loading script pane. You can use the button to locate the file.

3. Enter the `deliver` arguments.

   Note: Level defaults to 0, which is a good choice the first time you deliver your application. You will probably want to increase the Delivery level later, for reasons explained in the *LispWorks Delivery User Guide*.

4. Check the calculated Script Name (and modify it if desired), and click **OK**.

Figure 28.2 The New Delivery script dialog



The Application Builder now displays the path to the new script in its Build script pane. The new script will load patches, load your file, and then call **deliver**, something like this:

```
(in-package "CL-USER")
(load-all-patches)
(load "compile-and-load-my-app")
(deliver 'my-start-function "my-app" 0)
```

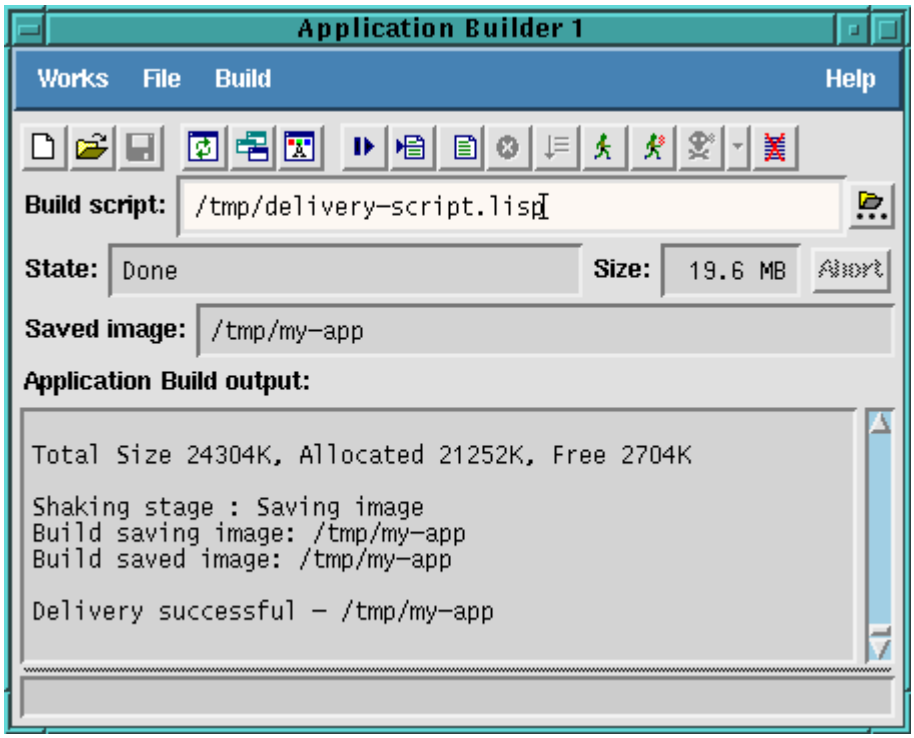
28.3 Building your application

Once you have a script name in the Build script pane, build your application by choosing **Build > Build** or clicking the toolbar button. The Application Builder invokes LispWorks in a subprocess, with the script as its **-build** argument.

If desired, you can abort the build process by pressing the **Abort** button.

The State pane displays the status of the building operation. After a successful build, the status changes to "Done" and the tool displays the name and size of the saved image in the Saved Image and Size panes, as shown in Figure 28.3 below.

Figure 28.3 The Application Builder after a successful build



28.4 Editing the script

The Application Builder makes it easy to find the script. Choose **Build > Edit Script** or click the toolbar button. Edit the script using the Editor tool that this displays. See “The Editor” on page 141 for more information about using the Editor tool.

Before it starts a build, the Application Builder saves the editor buffer displaying the script if you have modified that buffer. This behavior can be switched off - for the details, see “Configuring the Application Builder” on page 395.

28.5 Troubleshooting

During the build, the output is displayed in the Application Build output pane. This is a normal editor text box which you can search and edit in the usual way.

If there is an error during the build, a backtrace is generated and the subprocess image exits.

28.5.1 Viewing errors

To view the error message choose **Build > Display Error** or click the toolbar button.

To view the error message and the backtrace in an Editor tool choose **Build > Display Backtrace** or click the toolbar button. Most errors can be resolved after checking the backtrace.

28.5.2 Clearing the output

To clear the Application Build output pane choose **Build > Clear Output** or click the toolbar button.

You can set the tool to do this automatically - for the details see “Configuring the Application Builder” on page 395.

28.6 Running the saved application

Once you have successfully built your application, you can run it from the Application Builder.

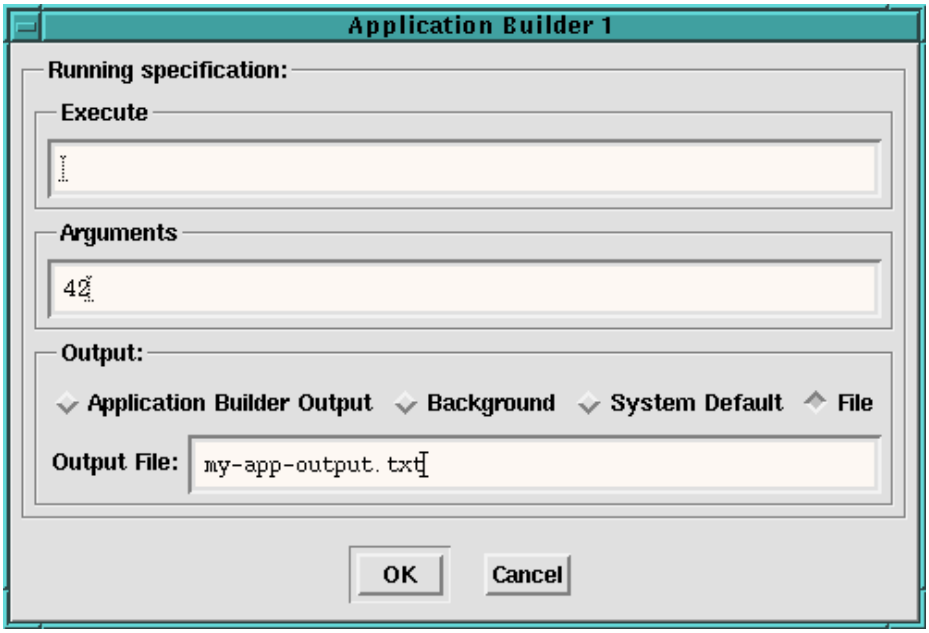
If the application can run without arguments you can run it by choosing **Build > Run** or clicking the toolbar button.

28.6.1 Passing arguments and redirecting output

If the application requires command line arguments, or you want to see what it writes to the standard output, or you need some other setups, choose **Build > Run With Argu-**

ments or click the toolbar button. This raises a dialog, shown in Figure 28.4 below.

Figure 28.4 The Run With Arguments dialog



To pass one or more command line arguments to your application, enter these in the Arguments pane.

To redirect the output of your application, select an option in the Output area.

Click **OK** to run your application with the settings you specified. The State pane shows when the application is running and reports when it has finished.

28.6.2 Executing a different file

The Run With Arguments dialog also allows you to set a different file to execute, rather than the saved image. This is useful if your application needs some setups, or if it needs to be invoked by some other program (for example, when it is a dynamic library).

To execute a different file from the one you built, enter the path in the Execute pane.

28.6.3 Killing application processes

Application processes that were invoked by the Application Builder can conveniently be killed if needed.

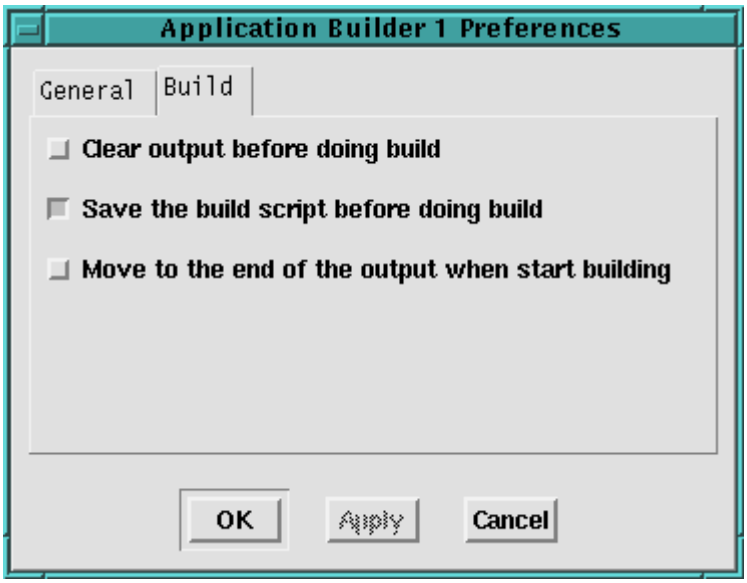
To kill all such processes, choose **Build > Kill All** or click the button.

To kill just one such process choose **Build > Kill Application** or click the dropdown to the right of the button. This raises a menu listing the invoked applications that are still running in the chronological order in which they were invoked. Select one item from the menu to kill that process.

28.7 Configuring the Application Builder

Using the Application Builder Preferences dialog, shown in Figure 28.5 below, you can configure the tool to best suit your needs. Choose **Works > Tools > Preferences...** or click to display this dialog.

Figure 28.5 Application Builder Preferences dialog



To prevent automatic saving of your edited script before a build, deselect the **Save the build script before doing build** option.

To make the Application Builder clear the output before each build, select the **Clear output before doing build** option.

To make the Application Builder ensure that the cursor is at the end of the current output before each build, select the **Move to the end of the output when start building** option.

You can control whether the Application Builder displays its toolbar by the option **Show Toolbar** on the General tab, as described in “Toolbar configurations” on page 24.

Index

Symbols

$ variable 237
***** variable **12**, 17, 233, 294
****** variable 12
******* variable 12
package 183
.lispworks file 27

A

aborting commands in the editor 169
accelerators
  for tools 21
action callbacks 283
Actions menu **35–37**
Alt key
  use of 145
application builder 387–396
Arguments command 185, 296
Attributes command 267, 269
Attributes menu 230
  Clip 234
  Copy 233
  Inspect 230

B

Backtrace command 298
backtraces 125
binding $ to the current inspector object 237
binding frames 130
Bindings button 130
Bindings command 298
Break command 314
Break on Access command 230
Break on Read command 230
Break On Return from Frame command 128
Break on Write command 230
breaking a process 314
breakpoints
  in the editor 181
Browse command **36**, 49, 381
  variations in name 36
Browse Metaclass command **81**, 84, 86, 90, 93, 95
Browse Parent command 385
Browse Parent System command **32**, 184, 366
Browse Representation command 385
Browse Screens command 385
Browse Symbols Like command 37, 297
browser-location variable 55
browsers 51
browsing
  Common Lisp classes 71–95
  compilation conditions 107–113
  errors 107
  function calls 187–195
  generic functions 197–206
  HTML documentation 51
  native window system 385
  online manuals 51
  output 11, **305–309**
  selected object, class of the 36, 296
  symbols 243–248
  systems 32, **366–378**
  window definitions 379–385
buffers
  closing 168

swapping between 168
Buffers menu 149, 178
Compile 180
Evaluate 179
Trace. *See* Trace menu
Undefine 184
bugs, reporting 298
Build command 388, 391
Build menu
Build 388, 391
Clear Output 393
Display Backtrace 393
Display Error 393
Edit Script 392
Kill All 395
Kill Application 395
Make a New Script 390
Run 393
Run With Arguments 393
building
applications 387–396
By Name option **39**
By Package option **39**

C

call frames 124, 129
callbacks
action 283
extend 283
retract 283
selection 283
specifying 282–284
catch frames 130
Catchers button 130
`:center` keyword 279
check components 261
choosing menu commands xiii
class browser 71–95
Class area 80
current class, operations specific to the 81, 84, 86, 90, 93, 95
description 3
Description area **83**, 86, 92
examining a class 77
Filter area 80
filtering information 75
Function description area 89
functions list 89
functions view 88–90
generic functions, operating on 90
Graph area 85
graph view 84–87
hierarchy view 77
Include Accessors button 89
Include Inherited button 89
inherited slots 74
Initargs area 92
initargs view 91
invoking on the current expression 296
invoking on the selected object 36
menu commands, see menu or command name
methods list 89
overview of the 71
Precedence area 95
precedence view 93
Slot description area 80
slot information 74–76
Slots area 80
slots view **74**–**76**
sorting information 78
tracing classes from the 90
undefining functions and methods 90
See also classes
Class command 15, **36**, 72, 75, 80, **296**
classes 71–95
changing slot values in the inspector 231–233
`column-layout` 256
displaying graphs of 84–87
examining 77
examining functions and methods defined on 88–90
inherited slots in 74
initargs 91
inspecting local slots 228
`interface` 251
`list-panel` 75
operations specific to the current class 81, 84, 86, 90, 93, 95
precedence list 93
`push-button-panel` 77
`row-layout` 256
tracing 90
See also class browser
Classes menu
Browse Metaclass **81**, 84, 86, 90, 93, 95
objects operated on by the 81, 83, 86, 90, 93, 95
Clear Output command 393
Clip command 100
clipboard
general use 28–29
interaction with UNIX clipboard 29

usage in editor 170
See also clipboard, kill ring, UNIX clipboard
Clone command **24**, 41, 169
Close command 168
interface builder 264
closing
editor buffers 168
Collapse Nodes command 60
collapsing graphs 59
`column-layout` class 256
Command to Key command 185, 303
commands
completion of 145
repeating 30
common features in the environment 19–50
See also under graphs
Common Lisp
classes. *See* classes
debugging 115–131
displaying documentation for expressions 296
displaying documentation for selected object 37, 127
evaluating forms 293–294
file extension 165
indentation of forms in source code 184
prompt 292
systems. *See* system
Common Lisp symbols 161
Common LispWorks podium 69–70
compilation conditions browser 107–113
pathnames 111
preference dialog 109
types to display 111
Compilation Options menu
Force 378
Preview 378
Source 378
Compile and Load command **32**, 125, 181, 369, 375
Compile and Load... command 181
Compile command **32**, 180, 181, 369, 375
Compile Region command 296
in editor 180
Compile... command 181
compiler output 180
`compile-system` function 364
compiling code
editor 180–181
compiling files in the listener 32
completion 46, 145
in class browser 77, 130
dynamic 162
in generic function browser 199
in-place 27, 162
Concatenate... command 369, 378
Condition menu
Actions. *See* Actions menu
Report Bug 127
confirmer
description 270
consistency in the environment. *See* common features in the environment
`contain` function 13, 75, 233
Contents radio button 54
Control key, use of **xiv**, 145
conventions used in the manual xi–xiv
Copy command **28**–**29**, 37, 170, 172, 232, 233
in Actions menu 329
interface builder 266
standard action command **37**
copying windows 24
creating new files **32**, 165
current
object. *See* selected object
package of any tool 34
prompt 293
value, operating on 297
current buffer 179
current class, operations specific to the 81, 84, 86, 90, 93, 95
current definition 179
current expression 179
displaying lambda list for 185, 296
displaying value 185, 296
stepper breakpoint 296
toggling stepper breakpoint 296
tracing 296
current form
macroexpanding 296
walking 297
Customize menu
Reusable 23
Cut command **28**–**29**, 170, 172
interface builder 255, 257, 262, **266**, 276

D

Debug command 314
Debug menu
Debugger 9, **115**, 116, 123, 125, 126
Listener 9
Restarts 298
debugger 115–131
abort restart 123

backtrace tree 120
binding frames 130
call frames 124, 129
catch frames 130
colors of variables 120
continue restart 123
controlling from the listener 297–298
debugger tool 298
description 122
displaying documentation for object in current frame 127
example session 124–127
finding source code for object in current frame 127
handler frames 130
invisible frames 130
invoking 117
invoking from the process browser 314
invoking from the tracer 40
menu commands in the listener 298
restart frames 130
restart options 123–124
stack 124
stack frames 124
See also debugger tool
Debugger command 9, **115**, 116, 123, 125, 298
debugger tool 115–131
Backtrace area 119
buttons 122
Condition area 119
invoking 115
invoking from notifier 118
package information 130
types of frame, displaying 129
See also debugger
debugging a process 314
defclass macro 89
Definitions menu 151, 152, 178
Compile 180
Evaluate 180
Generic Function 185
Trace. *See* Trace menu
Undefine 184
defsystem macro 212, 366
examples of use 365
deleting text in the editor 167–168
See also kill ring
Describe System
editor command 366
description
of compilation conditions 109
Description menu 109, 191, 200
Listen 247
Deselect All command **30**, 134, 170
Destroy command 385
Display as Confirmer command 270
Display as Dialog command 270
Display Backtrace command 393
Display Borders command 270, 271
Display Error command 393
display function 287
DISPLAY UNIX environment variable 5
displaying
package information 32–35
windows 20
display-message function 286
Documentation command 37, 127, 296
documentation, online. *See* online help
$ variable 237
dynamic library 394

E

Edit menu 28
Copy **28**, 37, 170, 172, 266
Cut **28**, 170, 172, 255, 257, 262, 266, 276
Deselect All **30**, 134, 170
Find 174, 177
Find Next **30**, 176
Find Next, for graph view 58
Find, for graph view 58
Find... **30**
interface builder 265
Link. *See* Link Menu
Paste **28**, 80, 170, 199, 266, 276
Replace 177
Replace... **30**
Search Files... 207
Select All **30**, 134, 170, 375
Undo **28**, 170
Edit Script command 392
Editing menu
Command to Key 185, 303
Key to Command 185, 303
editing the history list 31
editor
aborting commands 169
breakpoints 181
buffers view **147**, 168
buffers. *See* buffers
changed definitions list 151
closing buffers 168
compiling source code 180–181
creating a new window 169

creating files **32**, 165
current expression, displaying value 185
current package and displayed package 157
definitions list 150
definitions, operating on 184–185
deleting text 167–168
EMACS, comparison with 141
evaluating source code 179–180
expressions, operating on 184–185
History menu **168**, 169
indenting forms 184
inserting files into the current buffer 168
inserting text 167–168
invoking 142
keyboard commands, use of 145
kill ring. *See* kill ring
Lisp-specific commands 178–185
macroexpanding forms in the 185
menu and keyboard commands, distinctions 165
menu commands. *See* menu or command name
moving around in the buffer 166–167
new files **32**, 165
online help **185**, 303
opening files 32, 142, **165**
opening recent files 32
output view 146
overview 3
package information 157
package usage 183
repeating commands 170
replacing text 177–178
reverting to last saved version 165
saving files 165, 168
saving text regions 165
scrolling text 167
searching 174–177
sorting entries 156
swapping between buffers 168
tracing **182**
undefining symbols 184
undoing commands 170
using the clipboard 170
viewing two sections of the same file 169
views available 142
walking forms 185
editor commands
finding keyboard command for 303
`Shell` 336
`Visit Tags File` 146
EMACS 158
comparison with built-in editor 141
encoding 164
Enter Search String dialog 53
environment
common features 19–50
quitting 27
EOF command 336
error conditions 109
Escape key, use of **xiv**, 145
Evaluate command
in editor 179, 180
`Evaluate Last Form in Listener` editor command 303
Evaluate Region command
in editor 179
in listener 296
evaluating
code in the editor 179–180
forms 293–294
event
next 299
previous 299
repeating 31
examining objects 227
execute mode 298
Exit command 7, **27**
Expand Nodes command 60
expanding graphs 59
Expression menu 178
Arguments 185, 296
Browse Symbols Like 297
Class **296**
Compile Region 180, 296
Documentation 296
Evaluate Region 179, 296
Find Source 158, 296
Function Calls 297
Generic Function 297
Inspect Value 296
Macroexpand 185
Macroexpand Form 296
Toggle Breakpoint 296
Trace. *See* Trace menu
Value 185, 296
Walk 185
Walk Form 297
expressions
browsing the class of 296
displaying documentation 296
finding source code 296
extend callbacks 283
extended editor commands, finding keyboard

command for 303
:extended-selection keyword 279
external format 164

F

fg UNIX command 336
File menu 7, 70, 150
  Browse Parent System **32**, 184, 366
  Close 147, 168, 264
  Compile **32**, 181
  Compile and Load **32**, 125, 181, 182
  description 31–32
  Insert 168
  Load **32**, 180, 181, 182, 366
  New **32**, 165, 251, 275, 288
  Open **32**, 142, 146, 165
  Open... 252
  Print **32**, 166
  Recent Files **32**, 142
  Revert to Saved 165, 264
  Save 146, 150, 165, 264, 285, 288
  Save All 168, 264
  Save As 165
  Save As... 264
  Save Region As 165
filenames
  completion of 145
  extensions for CL files 165
files
  compiling in listener 32
  creating new **32**, 165
  inserting one into another 168
  loading 32
  navigating in the editor 166–167
  opening **32**, 142, 165
  opening recent **32**
  printing **32**
  reverting to last saved version 165
  saving 165
  saving all 168
filtering information **42**–**44**, 75, 374
  in inspector 225
filtering results 42
filters 42
Find command **30**
  in editor 177
  in graph view 58
Find Next command **30**, 176
  in graph view 58
Find Source command 37, 127, 142, 158, 296
  in Debug menu 298
  displaying list of results 158
  shortcut in debugger tool 126
Find... command **30**
  in editor 174
Force command 378
forms
  compiling in editor 180–181
  evaluating 293–294
  evaluating in editor 179–180
  indentation of 184
  re-evaluating 294–295
Frame menu
  Break On Return From Frame 128
  Documentation 127
  Find Source 126, 127, 142
  Method Combination 128, 201
  Restart Frame 128, 130
  Restart Frame Stepping 128, 340
  Return from Frame 128
  Trace. *See* Trace menu
function call browser 187–195
  By Name command 193
  By Package command 193
  Callees area 192
  Callers area 192
  description 187
  Documentation area 191
  Function area 188
  Function description area 191
  Function menu 195
    Trace submenu 195
  Graph area 189
  graphing callers and callees 189
  invoking on selected object 37, 297
  menu commands, see menu or command name
  operating on functions 195
  package information 194
  Show Package Names command 194
  sorting entries 193
  text view 191
  tracing from 195
  Unsorted command 193
  views available 187
Function Calls command 37
Function menu 199
  in the profiler 329
  Trace. *See* Trace menu
functions
  apropos 243
  compile-system 364
  contain 13, 75, 233

deliver 387
display 287
display-message 286
save-image 387, 389
undefining 90
Functions menu
in the class browser 90
in the function call browser 195

G

generic function browser 197–206
Arguments types area 204
description 197
Description area 200
displaying signatures 204
Filter area 199
Function area 199
invoking on selected object 37, 185
menu commands. *See* menu or command name
Method combination list 204
method combinations, viewing 201
methods list 200
operating on signatures 204
Signatures area 203
Generic Function command **37**, 185, 243, 297, 329
generic functions
browsing from listener 297
in class browser 90
defined on selected object 37, 185
get-inspector-values 238
global preferences
setting 25
When modified buffers 27
graph layout menu 57
Collapse Nodes 60
Expand Nodes 60
Preferences 63
Reset Graph Layout 61
graph view
system browser 367–369
graphical user interface. *See* interfaces
graphs 57–68
altering breadth 64
altering depth 64
children function 284, 287
different layouts 66–68
expanding and collapsing nodes 59
menu commands. *See* menu or command name
searching 58
sorting items 38
GUI. *See* interfaces

H

handler frames 130
Handlers button 130
Help menu 51, 70
Editing. *See* Editing menu
Lisp Knowledgebase 55
LispWorks Patches 55
Manuals 51
On Symbol 55
On Tool 54
Search 53
help. *See* online help
Hidden Symbols button 130
hierarchy view
in class browser 77
highlight
compiler messages 160
interactive input 160
matching parentheses 161, 178
selected text 160
history list 30
editing the 31
in the listener 295
repeating next event **31**, 299
repeating previous event **31**, 299
searching the 299
History menu 30
in editor **168**, 169
in the listener 295
interface builder 252
in listener 295
Modify 31
Next 31
Previous 31

I

Include Inherited Slots button 74, 80
Include Inherited Slots checkbox 16
incremental search 176
Index radio button 54
initargs of slot, displaying 81
initform of slot, displaying 81
initialization file 27
in-package 183
in-place completion 27
Insert... command 168
inserting files in editor 168
inserting text in editor 167–168

Inspect command 13, 25, **37**, 223, 229, 231, 246, 314
Inspect Value command 296
inspecting listener values automatically 237
inspector 223–241
  changing values 230–233
  description 224
  display options 234–236
  filtering display 225
  inspecting selected object 37
  menu commands. *See* menu or command name
  overview 3
  simple use 227
  sorting entries 234
  tracing 230
  tracing in the 230
  viewing local class slots 228
Inspector command 223
interface builder 249–271
  adding your own code 262
  attribute categories **268**–**269**
  Attributes dialog box 267, 278, 281
  button panels 255
  Check Component button 261
  code area 264
  code view **263**, 282
  Component button 260
  current interface 254
  current package 256
  default names of elements 257, 277
  default names of menus 259
  Edit menu 265
  editing code 263
  example of use 273–289
  interface area 254
  Interface menu 269–271
  interfaces, creating **251**–**252**, 275
  interfaces, loading **252**–**253**, 265
  introduction 249
  invoking 250
  Item button 259, 280
  layout hierarchy area **255**, 275
  layouts view **254**–**257**, 275–277
  Menu Bar button 259
  Menu button 259, 280
  menu hierarchy area **259**, 275
  menus view **258**–**262**, 280–282
  methods of use 262
  operating on elements 271
  Radio Component button 261
  rearranging components **266**, 276–277
  saving code 264–265, 284, 288
  setting attributes 267–269, 277–280
  switching between interfaces 252
  Update button 264
  views, description 250
  See also interfaces
`interface` class 251
Interface menu 49
  Attributes 269
  Display as Confirmer 270
  Display as Dialog 270
  Display Borders 270, 271
  interface builder 269–271
  Raise 270
  Regenerate 270
interface skeleton
  default menus in 252
  description 252
interfaces
  callbacks 269, 282–284
  confirmers 270
  constructing 275–277
  creating menus for 258–262
  creating new **251**–**252**, 275
  default package 256
  definition 251
  development strategy 262
  dialog boxes 270
  geometry of elements 269
  graph area 255
  layout elements, adding 256
  layout elements, removing 257
  layout hierarchy 255
  loading **252**–**253**, 265
  menu hierarchy 259
  menu objects, removing 262
  operating on the current 269–271
  rearranging components **266**, 276–277
  regenerating 270
  setting attributes 267–269, 277–280
  titles 268, 277–279
  types of attribute 268–269
interrupting evalution 295
interrupting Lisp 315
invisible frames 130
Invisible Functions button 130
`:items` keyword 75

K

KDE/Gnome emulation 158
key input 158
Key to Command command 185, 303

keyboard commands
  comparison with menu commands 165
  finding editor command for 303
  in the editor 145
keyboard conventions xiii–xiv
keywords
  :center 279
  :extended-selection 279
  :items 75
Kill All command 395
Kill Application menu 395
Kill command 314
kill ring 171–174
  copying text from 172–174
  marking the region 171
  putting text into 172
  rotating 173
  summary of use 174
killing a process 314

L

lambda list, displaying 185, 296
layouts
  adding to an interface 275–277
  pinboard 270
  rearranging **266**, 276–277
  specifying callbacks 282–284
  See also interfaces
layouts, displaying in window browser 383
Link from command 41
linking tools together 41
.lisp files 165
Lisp Knowledgebase command 55
LispWorks Patches command 55
Listen command **37**, 233, 246, 247, 314
listener
  basic tutorial 293–295
  browsing generic functions from 297
  compiling files in 32
  current expression, displaying value 296
  current expression, stepper breakpoint 296
  current expression, toggling stepper breakpoint 296
  debugger commands 300
  debugging in the 298
  description 292–293
  evaluating forms 293–294
  execute mode 298
  Expression menu. *See* Expression menu
  history commands 298
  history list 295
  History menu 295
  loading files in 32
  macroexpanding forms 296
  miscellaneous commands 301
  next event 299
  online help 303
  operating on expressions 295
  overview 3
  pasting selected object into 37
  previous event 299
  prompt 292
  re-evaluating forms 9, **294**–**295**
  searching history list 299
  size of the stack 303
  stack size 303
  *** variable 12
  ** variable 12
  * variable **12**, 17, 233, 294
  tracing current expression 296
  Values submenu. *See* Values menu
  walking forms 297
Listener Bind $ command 237
Listener command 292
list-panel class 75
Load command **32**, 180, 366, 369, 375
Load... command 180, 366
loading files 32
loading tools into the environment 21
local slots, inspecting 228
Long Names button 384
Lower command 266, 385

M

Macroexpand command 185
Macroexpand Form command 296
macros
  defclass 89
  defsystem 212, 366
  trace 40, 133
major tools, overview 2–4
Make a New Script command 390
manipulating values with inspector 230–233
Manuals command 51
manuals, online. *See* online help
menu commands
  check components 261
  choosing xiii
  comparison with keyboard commands in editor 165
  creating with the interface builder 258–262
  debugger commands 298
  names, specifying 282
  radio components 261

rearranging 266
specifying callbacks 284
See also interface
menu components 260–262
check 261
radio 261
menus
creating with the interface builder **258**–**262**, 280–282
rearranging 266
See also interface
Meta key
use of **xiv**
Method Combination command 128, 201
methods
displaying signatures 204
operating on signatures 204
undefining 90
viewing method combinations 201
Methods menu 89, 200
Trace. *See* Trace menu
Trace submenu. *See Trace menu*
Undefine 90
Modify command 31

N

native window representation 385
navigating within files in the editor 166–167
New command **32**, 165, 288
interface builder 251, 275
new files, creating **32**, 165
Next command
command line debugger 298
history list 31
next event
repeating **31**, 299

O

object clipboard
menu commands, see menu or command name
Object menu 35
Actions. *See* Actions menu
Attributes 267
Clip 234
Copy 233
interface builder 255, 256, **271**
Lower 266
Raise 266
objects
inspecting 227
operating on 35–37
searching for 30
selecting 30
See also selected object
On Symbol command 55
On Tool command 54
online help 51–56
browsing manuals 51
current symbol 55
current tool 54
packages, searching 54
searching 53–54
Open command **32**, 142, 165, 168
Open... command
interface builder 252
opening files **32**, 142, 165
opening recent files **32**
operating on objects 35–37
See also objects
Operations menu
Break 336
EOF 336
Suspend 336
output
compiler 180
editor 146
standard **305**–**309**
output browser 11, **305**–**309**
menu commands. *See* menu or command name
overview 3
overview of major tools 2–4
overview of profiling 323

P

Package command
interface builder 256
packages
current package 34
display of 32–35
in editor 183
searching for documentation 54
Packages button 333
Packages... button 326, 334
Page Down key 167
Page Up key 167
Parent command 368, **369**
Partial Search radio button 54
Paste command **28**–**29**, 170
in class browser 80
in generic function browser 199
in inspector **231**, 232, 233

interface builder **266**, 276
pinboard objects
moving and resizing 270
podium. *See* Common LispWorks podium
Preferences command 24, 33, 34, 39, 49, 63, 78, 80, 89, 109, 129, 155, 216, 302
Preferences... command 63, 206, 231, 248, 315, 328, 333, 358, 377, 382, 395
Preview command 378
previewing a system plan **372**–**375**, 378
Previous command
command line debugger 298
history list 31
previous event
repeating **31**, 299
primary package in editor 183
Print command **32**
Print... command 166
printing files **32**
process browser 311–316
menu commands. *See* menu or command name
sorting processes 314
processes
breaking 315
inspecting 314
killing 314
sorting 313
Processes menu 314
profiler 317–334
choosing packages 326–328
choosing symbols 324–328
description 319–320
example of use 332–334
information returned 323, **329**
interpreting results 329
menu commands. *See* menu or command name
overview of profiling 323
pitfalls 330
running a profile 328
sorting results 329
specifying code to run 328
symbols that can be profiled 330
prompt in the listener 292
`push-button-panel` class 77

Q

Quit command 385
quitting the environment 27

R

radio components 261
Raise command 266, 270, 385
readers of a slot, displaying 81
Recent Files command **32**, 142
Recompute Events button 374
recursive macroexpansion 297
re-evaluating forms in listener 9, **294**–**295**
Refresh command 14, **25**
Regenerate command 270
regexp 44
syntax 44
regular expressions 44
syntax 44
Remote Shell tool 337
repeating commands 30
in the editor 170
repeating the next event 31
repeating the previous event 31
Replace command **30**, 177, 370
Replace... command **30**, 177
replacing text 177–178
Report Bug command 127, 298
reporting bugs 127, 298
Restart Frame command 128, 130
Restart Frame Stepping command 128, 340
restart frames 130
Restarts button 130
Restarts menu 123, 350
Restarts submenu 298
retract callbacks 283
Return from Frame command 128
re-using windows 23
Revert to Saved command 165
interface builder 264
reverting a file to the version stored on disk 165
`row-layout` class 256
Run command 393
Run With Arguments command 393

S

Save All command
interface builder 264
Save All... command 168
Save As... command **165**
interface builder 264
Save command **165**, 288
interface builder 264, 285
Save Region As... command 165
saving all files 168
saving files 165

interface builder 264–265
saving regions of text 165
scrolling text in editor 167
Search command 53
Search Files... command 370
searching
for objects 30
for text 30, **174**–**177**
history list 299
online manuals 53
Select All command **30**, 134, 170, 375
selected object
browsing 36
browsing the class of 36
copying 37
displaying documentation 37
finding source code 37
inspecting 37
pasting into listener 37
placing on object clipboard 36
showing function calls 37, 297
showing generic functions 37, 185
showing similar symbols 37
selection callbacks 283
Set command **231**, 231
debugger 129
`Shell` editor command 336
Shell tool 335
shell tool 335–337
break signal, sending 336
creating 335
EOF signal, sending 336
menu commands. *See menu* or command name
recalling commands 337
suspend signal, sending 336
type of shell 337
`*shell-shell*` variable 337
Shift key, use of **xiv**, 145
Short Names button 384
Show in Tracer command 41, 133
Show Package Names button **34**, 130, 235, 383
Show Toolbar button **24**
Signature menu 204–206
signatures
displaying 204
operating on 204
`simple-pane-foreground` reader 17
Slots menu 35, 80, 92
Clip 234
Copy 233
Inspect 229
Paste **231**, 232, 233
Set **231**, 231
sort options
By Name 39
By Package 39
Unsorted 40
sorting
in class browser 78
in editor 156
in inspector 234
in process browser 314
views 38
in window browser 383
source code
debugging 115–131
for current expression 296
for object in current frame of debugger 127
for selected object 37
Source command 378
stack frames in the debugger 124
standard action command
Browse **36**
Browse Symbols Like **243**, **297**
Class **36**
Copy **37**
Documentation **37**
Find Source **37**
Function Calls **297**
Generic Function **37**, **185**, **297**
Inspect **37**
Listen **37**
standard output **305**–**309**
`*standard-output*` variable 11
`***` variable 12
`**` variable 12
`*` variable **12**, 17, 233, 294
stepping through code 339–361
Stop command 314
stopping a process 314
Suspend command 336
swapping editor buffers 168
Symbol Browser 243–248
symbol browser
invoking on selected object 37
Symbol menu
Inspect 246
Listen 246
Unintern... 246
symbols
interface builder 271
online help for 55

tracing 182
undefining 184
Symbols... button 325, 334
syntax coloring 161, 178
system
browsing 366
compiling and loading 369
concatenating 369
creating plans for 374
defining 288, **364–366**
executing plans for 374
forcing compilation and loading of members 378
introduction to 363–364
parent system, browsing 368
plan 372
previewing a plan **372–375**, 378
`ROOT-SYSTEM 368`, 372
searching 370
using source files 378
system browser 363–378
Actions area 374
compiling and loading systems 369
creating plans 374
description 366
executing plans 374
File description area 369
Filter area 374
forcing compilation and loading 378
Graph area 368
graph view 367–369
menu commands. *See* menu or command name
output view 375–376
package information 377
parent system, browsing 368
Plan area 374
previewing the plan 372–375
sorting information 377
System area 368
system plan, previewing **372–375**, 378
text view 370–372
using 366–369
using source files 378
views available 366
Systems menu 369
Compile **369**, 375
Compile and Load **369**, 375
Concatenate... 369
Hide Files 370
Load **369**, 375
Parent 368, **369**
Replace 370
Search Files 370
Show Files 370

T

tab completion
interface builder 254
text
deleting 167–168
inserting 167–168
replacing 177–178
saving regions of 165
scrolling in editor 167
searching for 30, **174–177**
selecting 30
See also under editor
text view
in editor 143
in function call browser 191
in system browser 370–372
Toggle Breakpoint command 296
Toggle Tracing command 41
toolbars
hiding 24
removing 24
tools
current package of 34
linking together 41
loading into the environment 21
online help for 54
overview of major 2–4
reusing 26
tracing from 40–41
Tools menu 2, 7, 21, 70
accelerators 21
Application Builder 387
Class Browser 72, 98, 107, 142, 197
Editor **142**
Global Preferences 23, 25, 27, 47
Inspector 223
Interface Builder 250
Listener **292**
Object Clipboard 98, 107, 187
Output Browser **306**
Preferences 24, 33, 34, 39, 49, 63, 78, 80, 89, 109, 129, 155, 216, 302
Preferences... 206, 231, 248, 315, 328, 333, 358, 377, 382, 395
Process Browser 312
Profiler 317
Search Files 207
Shell 335

Stepper 339, 342
Symbol Browser 243
System Browser 366
Tracer 133
Window Browser 379
Trace command **40**, 182, 297
Trace Inside command 40
trace macro 40, 133
Trace menu
Break on Access 230
Break on Read 230
Break on Write 230
Show in Tracer 41, 133
Trace 40
Trace Inside 40
Trace Read 230
Trace with Break 40
Tracing 41
Untrace 40, 230
Untrace All 41
Trace with Break command 40
Tracer 133–139
Function menu 134
tracing 133–139
classes 90
in function call browser 195
in the inspector 230
in inspector 230

U

Undefine command 90, 184
Undefine... command 184, 201
undefining
current definition 184
functions 90
generic functions 201
methods 90
Undo command 28
editor 170
Unintern... command 246
UNIX clipboard 80, 199
interaction with Common LispWorks clipboard 29
usage in editor 172
Unsorted option **40**
Unstop command 314
unstopping a process 314
Untrace All command 41
Untrace command 40, 230
updating windows 25
using the clipboard 28–29
See also kill ring
using the keyboard xiii–xiv
using the mouse xi–xiii

V

Value command 185, 296
Value menu
Listen 233
values
changing in inspector 230–233
Values menu
Class 15, 72, 75
Copy 232, 233
Inspect 25, 223, 229, 231
variables
***** 301
$ 237
*** 12**, 17, 233, 294
****** 12
******* 12
browser-location 55
grep-command 221
grep-command-format 221
grep-fixed-args 221
packages-for-warn-on-redefinition 327
shell-shell 337
standard-output 11
trace-verbose 137
Variables menu
Set 129
views
in class browser 71
description 37–40
in editor 142
in function call browser 187
in generic function browser 197
graph **57**–**68**, 367–369
hierarchy 77
in inspector 234
output 146, **305**–**309**, 375–376
slots **74**–**76**
sorting items in 38
in system browser 366
text 191, 370–372
Visit Tags File editor command 146

W

Walk command 185
Walk Form command 297
web browsers 51
Whole Word radio button 54

window browser 379–385
  changing root of graph 381
  complete window names, displaying 384
  destroying a window 385
  different types of window 383
  lowering a window 385
  menu commands. *See* menu or command name
  moving around different windows 385
  native representation 385
  package information 383
  quitting a window 385
  raising a window 385
  sorting entries 383
  using 381
  whole environment 385
window colors 162
windows
  displaying 20
  making copies of 24
  re-using 23
  updating 25
Windows menu 14, **20**, 70, 385
  Actions. *See* Actions menu
  Browse 381
  Browse Parent 385
  Browse Representation 385
  Browse Screen 385
  Destroy 385
  Lower 385
  Quit 385
  Raise 385
  in window browser 385
Works menu 70
  Clone **24**, 41
  Exit 7, **27**
  Exit Window 147
  Object submenu. *See* Object menu
  Packages submenu. *See* Packages menu
  Symbols submenu. *See* Symbols menu
  See also individual entries for each submenu
writers for a slot, displaying 81