

# 4 Program Development Utilities

*symbolics*



# 4 Program Development Utilities

*symbolics™*

---

# Program Development Utilities

# 999021

August 1986

**This document corresponds to Genera 7.0 and later releases.**

The software, data, and information contained herein are proprietary to, and comprise valuable trade secrets of, Symbolics, Inc. They are given in confidence by Symbolics pursuant to a written license agreement, and may be used, copied, transmitted, and stored only in accordance with the terms of such license. This document may not be reproduced in whole or in part without the prior written consent of Symbolics, Inc.

Copyright © 1986, 1985, 1984, 1983, 1982, 1981, 1980 Symbolics, Inc. All Rights Reserved.

Portions of font library Copyright © 1984 Bitstream Inc. All Rights Reserved.

Portions Copyright © 1980 Massachusetts Institute of Technology. All Rights Reserved.

**Symbolics, Symbolics 3600, Symbolics 3670, Symbolics 3675, Symbolics 3640, Symbolics 3645, Symbolics 3610, Symbolics 3620, Symbolics 3650, Genera, Symbolics-Lisp<sup>®</sup>, Wheels, Symbolics Common Lisp, Zetallisp<sup>®</sup>, Dynamic Windows, Document Examiner, Showcase, SmartStore, SemantiCue, Frame-Up, Firewall, S-DYNAMICS<sup>®</sup>, S-GEOMETRY, S-PAINT, S-RENDER<sup>®</sup>, MACSYMA, COMMON LISP MACSYMA, CL-MACSYMA, LISP MACHINE MACSYMA, MACSYMA Newsletter and Your Next Step in Computing** are trademarks of Symbolics, Inc.

## Restricted Rights Legend

Use, duplication, and disclosure by the Government are subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software Clause at FAR 52.227-7013.

Symbolics, Inc.  
4 New England Tech Center  
555 Virginia Road  
Concord, MA 01742

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.

Text masters produced on Symbolics 3600-family computers and printed on Symbolics LGP2 Laser Graphics Printers.

Cover Design: Schafer|LaCasse

Printer: CSA Press

Printed in the United States of America.

Printing year and number: 88 87 86 9 8 7 6 5 4 3 2 1

## Table of Contents

	Page
<b>I. Debugger</b>	<b>3</b>
<b>1. Overview of the Debugger</b>	<b>5</b>
1.1 Overview of Debugger Commands	7
1.2 Overview of Debugger Evaluation Environment	9
1.3 Overview of Debugger Mouse Capabilities	10
1.4 Overview of Debugger Help Facilities	10
<b>2. Entering and Exiting the Debugger</b>	<b>11</b>
2.1 Entering the Debugger	11
2.1.1 Entering the Debugger by Causing an Error	11
2.1.2 Entering the Debugger With <code>m-SUSPEND</code> , <code>c-m-SUSPEND</code>	13
2.1.3 Entering the Debugger With <code>break</code> And <code>zl:dbg</code> Functions	14
2.2 Exiting the Debugger	15
<b>3. Using the Debugger</b>	<b>17</b>
3.1 Entering a Debugger Command	17
3.1.1 Debugger Command Accelerators	18
3.1.2 Editing a Debugger Command	19
3.1.3 Entering a Debugger Command with the Mouse	19
3.2 Getting Help for Debugger Commands	19
3.3 Proceeding and Restarting in the Debugger	20
3.3.1 Using Debugger Proceed and Restart Options	20
3.4 Evaluating a Form in the Debugger	22
3.4.1 Editing a Form in the Debugger	23
3.4.2 Rebound Variable Bindings During Evaluation	24
3.5 Using Recursive Debugger Invocations	25
3.6 Using the Mouse in the Debugger	27
3.7 Debugger Command Descriptions	29
3.7.1 Debugger Commands for Viewing a Stack Frame	31
3.7.2 Debugger Commands for Stack Motion	37
3.7.3 Debugger Commands for General Information Display	41
3.7.4 Debugger Commands to Continue Execution	50
3.7.5 Debugger Trap Commands	52
3.7.6 Debugger Commands for Breakpoints and Single Stepping	57



3.7.7	Debugger Commands for System Transfer	61
3.7.8	Miscellaneous Debugger Commands	63
3.8	Debugger Functions	63
3.9	Debugger Variables	66
<b>4.</b>	<b>Summary of Debugger Commands</b>	<b>67</b>
<b>5.</b>	<b>Tracing Function Execution</b>	<b>73</b>
5.1	Options To trace	74
5.2	Controlling the Format Of trace Output	77
5.3	Untracing Function Execution	78
<b>6.</b>	<b>Advising a Function</b>	<b>79</b>
6.1	Designing the Advice	82
6.2	:around Advice	82
6.3	Advising One Function Within Another	83
<b>7.</b>	<b>Stepping Through an Evaluation</b>	<b>85</b>
<b>8.</b>	<b>evalhook</b>	<b>87</b>
8.1	applyhook	88
<b>II.</b>	<b>Miscellaneous Debugging Aids</b>	<b>91</b>
<b>9.</b>	<b>The Inspector</b>	<b>93</b>
9.1	How the Inspector Works	93
9.2	Entering and Leaving the Inspector	93
9.3	The Inspector Interaction Pane	95
9.4	The Inspector History Pane	95
9.5	The Inspector Menu Pane	95
9.6	The Inspector Inspection Pane	96
9.6.1	Inspection Pane Display	97
9.7	Special Characters Recognized by the Inspector	97
9.8	Examining a Compiled Code File	98
<b>10.</b>	<b>The Peek Program</b>	<b>99</b>
10.1	Overview of Peek	99
10.2	Peek Modes	101
<b>III.</b>	<b>The Compiler</b>	<b>105</b>

<b>11. Introduction to the Compiler</b>	<b>107</b>
11.1 How to Invoke the Compiler	107
<b>12. Structure of the Compiler</b>	<b>109</b>
12.1 How the Stream Compiler Handles Top-level Forms	111
12.1.1 Controlling the Evaluation of Top-level Forms	115
12.2 Function Compiler	116
12.3 Bin File Dumper	117
12.4 Compiler Tools and Their Differences	117
12.4.1 Tools for Compiling Code From the Editor Into Your World	117
12.4.2 Tools for Compiling Files	118
12.4.3 Tools for Compiling Single Functions	120
<b>13. Compiler Warnings Database</b>	<b>123</b>
<b>14. Controlling Compiler Warnings</b>	<b>127</b>
14.1 Compiler Style Warnings	127
14.2 Function-referenced-but-never-defined Warnings	130
14.2.1 Overriding Variable-defined-but-never-referenced Warnings	131
<b>15. Compiler Switches</b>	<b>133</b>
<b>16. Compiler Source-Level Optimizers</b>	<b>135</b>
<b>17. Files That Maclisp Must Compile</b>	<b>137</b>
<b>18. Putting Data in Compiled Code Files</b>	<b>139</b>
<b>IV. Maintaining Large Programs</b>	<b>141</b>
<b>19. Introduction to the System Construction Tool</b>	<b>143</b>
<b>20. Defining a System</b>	<b>145</b>
20.0.1 defsystem Options	146
20.0.2 defsystem Modules	153
20.1 defsystem Operations	168
20.1.1 Table of Module Types and Operations	169
20.1.2 System Plan	171
20.2 User-defined Module Types	171
20.3 User-defined Operations on Systems	172

<b>21. Loading and Compiling Systems</b>	<b>175</b>
21.0.1 load-system Keywords	176
21.0.2 compile-system Keywords	179
21.1 Loading System Definitions That Use Logical Pathnames	180
21.1.1 Sys:site;System-name.System File	181
21.1.2 Sys:site;Logical-host.Translations File	182
21.1.3 System Declaration File	183
21.2 Loading System Definitions That Use Physical Pathnames	185
<b>22. Other Operations on Systems</b>	<b>187</b>
22.1 Editing, Hardcopying, Reap-Protecting, and Releasing Systems	187
<b>23. Directories Associated with a System</b>	<b>191</b>
23.1 Component Directory File	193
23.2 Contents of the Patch Directory Files	194
<b>24. Patch Facility</b>	<b>197</b>
24.1 Types of Patch Files	199
24.1.1 Patch Directory File	200
24.1.2 Individual Patch Files	200
24.1.3 Organization of Patch Files	200
24.1.4 Names of Patch Files	201
24.2 Making Patches	203
24.2.1 Start Patch (m-X)	205
24.2.2 Start Private Patch (m-X)	205
24.2.3 Add Patch (m-X)	206
24.2.4 Add Patch Changed Definitions of Buffer (m-X)	206
24.2.5 Add Patch Changed Definitions (m-X)	207
24.2.6 Select Patch (m-X)	207
24.2.7 Show Patches (m-X)	207
24.2.8 Finish Patch (m-X)	208
24.2.9 Abort Patch (m-X)	209
24.2.10 Resume Patch (m-X)	209
24.2.11 Recompile Patch (m-X)	209
24.2.12 Reload Patch (m-X)	209
24.3 Loading Patches	210
<b>25. Obtaining Information About a System</b>	<b>213</b>
25.1 Obtaining Information on System Versions	215

<b>V. Program Counter Metering</b>	<b>219</b>
<b>26. PC Metering</b>	<b>221</b>
<b>VI. Program Development Tools and Techniques</b>	<b>225</b>
<b>27. Introduction</b>	<b>227</b>
27.1 Purpose	227
27.2 Prerequisites	227
27.3 Scope	227
27.4 Method	227
27.5 Features	228
27.6 Organization	228
<b>28. Writing and Editing Code</b>	<b>231</b>
28.1 Using Zmacs	231
28.1.1 Using The HELP Key in Zmacs	231
28.1.2 Zmacs Command Completion	232
28.2 Preparing to Write Code	233
28.2.1 Entering the Editor	233
28.2.2 Creating a New File	234
28.2.3 Creating a File Attribute List	234
28.2.4 Major and Minor Modes	236
28.3 Program Development: Design and Figure Outline	238
28.3.1 Program Strategy	238
28.3.2 Simple Screen Output	239
28.3.3 Outlining the Figure	240
28.4 Keeping Track of Lisp Syntax	247
28.4.1 Comments	249
28.4.2 Aligning Code	251
28.4.3 Balancing Parentheses	252
28.5 Program Development: Drawing Stripes	252
28.6 Finding Out About Existing Code	260
28.6.1 Finding Out About Objects	260
28.6.2 Finding Out About Symbols	263
28.6.3 Finding Out About Variables	265
28.6.4 Finding Out About Functions	265
28.6.5 Finding Out About Pathnames	271
28.7 Program Development: Refining Stripe Density and Spacing	272
28.8 Editing Code	282
28.8.1 Identifying Changed Code	282
28.8.2 Searching and Replacing	284
28.8.3 Moving Text	286
28.8.4 Keyboard Macros	292

28.8.5	Using Multiple Windows	293
<b>29.</b>	<b>Compiling and Evaluating Lisp</b>	<b>297</b>
29.1	Compiling Lisp Code	298
29.1.1	Compiling Code in a Zmacs Buffer	299
29.1.2	Compiling and Loading a File	301
29.2	Evaluating Lisp Code	303
29.2.1	Evaluation and the Editor	303
29.2.2	Lisp Input Editing	306
<b>30.</b>	<b>Debugging Lisp Programs</b>	<b>309</b>
30.1	Using the Compiler Warnings Database	309
30.2	Using the Debugger	310
30.3	Commenting Out Code	314
30.4	Tracing and Stepping	323
30.4.1	Tracing	323
30.4.2	Stepping	325
30.5	Using Breakpoints	329
30.6	Expanding Macros	332
30.7	Using the Inspector	335
<b>31.</b>	<b>Using Flavors and Windows</b>	<b>343</b>
31.1	Program Development: Modifying the Output Module	344
31.1.1	A Mixin to Position the Figure	345
31.1.2	The Basic Arrow Window	349
31.1.3	Converting LGP to Screen Coordinates	354
31.1.4	Flavors for LGP Output	357
31.1.5	The Top-Level Function	359
31.1.6	The Arrow Window: Interaction, Processes, and the Mouse	364
31.1.7	Defining Flavors to Signal Conditions	369
31.2	Programming Aids for Flavors and Windows	377
31.2.1	General Information on Flavors	377
31.2.2	Methods	378
31.2.3	Init Keywords	381
<b>32.</b>	<b>Calculation Module for the Sample Program</b>	<b>383</b>
<b>33.</b>	<b>Output Module for the Sample Program</b>	<b>403</b>
<b>34.</b>	<b>Graphic Output of the Sample Program</b>	<b>425</b>
<b>Index</b>		<b>427</b>

## List of Figures

Figure 1.	The Inspector	94
Figure 2.	Program output with only the outlines of the arrows in the figure.	248
Figure 3.	Program output with stripes of even spacing and density.	261
Figure 4.	Program output with stripes of varying spacing and density.	283
Figure 5.	Using multiple windows to test the program while viewing the source code.	296
Figure 6.	Edit Compiler Warnings (m-x) splits the screen. The upper window contains compiler warnings. The lower window contains the source code.	311
Figure 7.	The Window Debugger: inspecting the stack frame containing a call to <b>compute-dens</b> .	315
Figure 8.	The Window Debugger: inspecting the variable <b>*x2*</b> .	316
Figure 9.	Output resulting from a faulty attempt to outline the small arrows recursively.	320
Figure 10.	Output resulting from a faulty attempt to outline the small arrows recursively, with the second function call commented out.	321
Figure 11.	Output resulting from a corrected attempt to outline the small arrows recursively, with the second function call commented out.	322
Figure 12.	Output from the program with a bug in the function <b>draw-arrow-shaft-stripes</b> .	333
Figure 13.	The Inspector window: inspecting an instance of a structure.	338
Figure 14.	The Inspector window: inspecting an instance of a flavor.	339

## List of Tables

Table 1.	Trace Menu Items and <b>trace</b> Options	326
----------	---	-----



## Introduction to Program Development Utilities

This document describes program development utilities available to you in Genera, the Symbolics software environment. These utilities include:

- The Genera interactive Debugger. See the section "Debugger", page 3.
- Debugging utilities: Trace, Advise, Step, and Evalhook. See the section "Debugger", page 3.
- The Inspector and Peek utilities. See the section "Miscellaneous Debugging Aids", page 91.
- The compiler. See the section "The Compiler", page 105.
- Utilities for maintaining large programs. See the section "Maintaining Large Programs", page 141.
- Metering utilities. See the section "Program Counter Metering", page 219.

This document also suggests some programming style and techniques for developing programs in the Genera software environment. It describes ways of using Genera features that might help you through various stages of the program development process. See the section "Program Development Tools and Techniques", page 225.





**PART I.**

**Debugger**



## 1. Overview of the Debugger

Genera, the Symbolics software environment offers you a host of powerful debugging tools. The most comprehensive of these tools is the Symbolics interactive Debugger and its window-oriented counterpart, the Window Debugger.

Other debugging tools are:

- The *Trace* facility, which performs certain debugging actions when a function is called or when a function returns. See the section "Tracing Function Execution", page 73.
- The *Advise* facility, which modifies the behavior of a function. See the section "Advising a Function", page 79.
- The *Step* facility, which allows you to execute forms in your program, one at a time, so that you can examine what is happening when execution suspends at every "step." See the section "Stepping Through an Evaluation", page 85. The Debugger's `:Single Step` command also performs stepping. See the section "Single Step Command", page 61.
- The *evalhook* facility, which allows you to get a particular Lisp form whenever the evaluator is called. The Step facility also uses evalhook. See the section "evalhook", page 87.

Two other tools related to debugging are the *Inspector* and *Peek*. The Inspector is a window-oriented program that lets you inspect data objects and their components. Peek is a program that gives a dynamic display of various kinds of system status. For information about the Inspector: See the section "The Inspector", page 93. For information about Peek: See the section "The Peek Program", page 99.

For information on the Window Debugger: See the section "Debugging Lisp Programs", page 309.

In the Genera software environment, unlike more traditional programming environments, you do not have to include the Debugger explicitly when you compile your programs. Generally, you can debug your code as you write it without having to perform a series of complicated compiling, loading, and executing procedures between source code development and debugging.

Because Symbolics user-interface features allow you to perform many Symbolics activities simultaneously – Zmacs, Zmail, the file system, a Dynamic Lisp Listener, and so on – debugging becomes an easy task, regardless of how many system activities you are using. You can move in and out of the Debugger as easily as you can move in and out of any other activity in Genera.

For example, the Debugger command, `:Edit Function`, brings up a specified function for you to edit in a Zmacs editor window. This is useful when you have found the function that caused the error and want to edit that function immediately. Another command, `:Mail Bug Report`, creates a bug report message in a mail window and puts a backtrace into it. While composing the bug report, you can switch back and forth between the Debugger and the mail window.

The Symbolics Debugger is there whenever you need it. The Debugger is invoked whenever an error occurs in your program's execution or the execution of a system function. That is, your machine brings you into the Debugger whenever it encounters an error that is not handled by a condition handler, for example, when you reference an unbound variable. See the section "Entering and Exiting the Debugger", page 11. Once in the Debugger, you are given a choice of actions that can correct the error. These actions are called *proceed* and *restart options*. See the section "Proceeding and Restarting in the Debugger", page 20.

You can also enter the Debugger explicitly, at any time, by pressing `m-SUSPEND` or `c-m-SUSPEND`. Or you can make your program enter the Debugger by inserting the `break` or `zl:dbg` function into your program code. See the section "Entering and Exiting the Debugger", page 11.

Upon Debugger entry, besides selecting one of the *proceed* and *restart options*, you can enter any of the Debugger's commands. These commands are full-form English commands, built on the normal Command Processor (CP) substrate. In fact, several Debugger commands are in the global command table. For more information on Debugger commands: See the section "Entering a Debugger Command", page 17. Also: See the section "Debugger Command Descriptions", page 29.

In the Debugger you can also evaluate a form in the lexical (user-program) context of the current frame. This context is referred to as the Debugger's *evaluation environment*. You can think of the Debugger's evaluation environment as a special read-eval-print loop that not only evaluates forms but also evaluates them in the context of the suspended function, where the lexically apparent values of all the local variables are accessible. For more information on the evaluation environment: See the section "Evaluating a Form in the Debugger", page 22.

Like other output in the Genera software environment, Debugger output is mouse sensitive, so you can perform many useful Debugger operations using the mouse. For more information on mouse capabilities: See the section "Using the Mouse in the Debugger", page 27.

The Debugger also provides some online help facilities. For more information on help facilities: See the section "Getting Help for Debugger Commands", page 19.

For complete information on the uses of these features and other Debugger features – plus a list of descriptions for all Debugger commands: See the section "Using the Debugger", page 17.

In general, you would use the Debugger when:

- Your program triggers the Debugger because garbage – an unbound variable or too many arguments perhaps – was passed to a function, and you want to find out where the garbage came from. See the section "Analyze Frame Command", page 43.
- You want to see what's happening in the sequence of function calls just executed, including a history of these function calls, the argument values passed, the local-variable values, the source code, and the compiled code. See the section "Show Backtrace Command", page 45. Also: See the section "Debugger Commands for Viewing a Stack Frame", page 31.
- You want to find out who or what is referencing a special variable or any other location in memory. See the section "Monitor Variable Command", page 54.
- You want to perform debugging operations using the mouse. See the section "Using the Mouse in the Debugger", page 27.
- You want to continue program execution, proceed from an error, restart a function, return from a function, or throw through a function. See the section "Debugger Commands to Continue Execution", page 50.
- Your condition handler does not work properly, and you want to debug this handler when it is encountered. See the section "Enable Condition Tracing Command", page 54.
- You want to edit your function's source code in Zmacs immediately after you have found the error. See the section "Edit Function Command", page 61.
- You want to put a Debugger backtrace into a mail message and send this message as a bug report. See the section "Mail Bug Report Command", page 62.
- You want to use Debugger breakpoint commands, instead of using the Trace facility or inserting a function in your code, to set Debugger breakpoints. See the section "Commands for Breakpoints and Single Stepping".

## 1.1 Overview of Debugger Commands

The Debugger comprises more than 50 full-form English commands, which are implemented as CP commands. Debugger commands are entered inside the

Debugger at the Debugger's command prompt, a right arrow (→). Commands fall into eight general categories:

Commands for viewing a stack frame

Commands for stack motion

Commands for general information display

Commands to continue execution

Trap commands

Commands for breakpoints and single stepping

Commands for system transfer

Miscellaneous commands

Most Debugger commands have corresponding key-binding accelerators, which means you can press a combination of one or more keys in place of the command. For example, you can press the accelerator `c-E` instead of the command `:Edit Function`.

Most Debugger commands also have keywords you can use to modify the command's behavior.

Many Debugger commands share the global command table. Therefore, you can enter these commands while you are in a CP command loop. You do not have to be in the Debugger. These commands are:

:Clear All Breakpoints  
:Clear Breakpoint  
:Disable Condition Tracing  
:Edit Function  
:Enable Condition Tracing  
:Monitor Variable  
:Set Breakpoint  
:Set Stack Size  
:Show Breakpoints  
:Show Compiled Code  
:Show Monitored Locations  
:Show Source Code  
:Unmonitor Variable

Note, however, that you must type a preceding colon with every command entered in the Debugger; for example, you must type :Set Breakpoint in the Debugger.

For complete information on Debugger commands: See the section "Entering a Debugger Command", page 17.

## 1.2 Overview of Debugger Evaluation Environment

In the Debugger, you can evaluate a form as easily as you can in a Dynamic Lisp Listener read-eval-print loop. Evaluating a form in the Debugger, however, is particularly useful because you are evaluating the form in the context of a user program and the current stack frame. This means you can see the value of Lisp objects at the point in program execution where an error occurred or at the precise place in your program where you explicitly suspend execution and invoke the Debugger. You can even reference lexical (local) variables at the point where execution suspends.

Evaluating a form in the Debugger is a simple task. If you type a character other

1  
6



than the first character in a Debugger command – a colon or accelerator key – the Debugger immediately brings you into its evaluation environment. In other words, just type the form. Evaluation happens automatically.

For complete information on how to evaluate a form in the Debugger: See the section "Evaluating a Form in the Debugger", page 22.

### 1.3 Overview of Debugger Mouse Capabilities

When the output generated by Debugger commands is displayed in a Dynamic Window, it is mouse sensitive. You can perform several useful debugging operations simply by using the mouse to click on something. Some of these operations include: setting a breakpoint, monitoring a variable or another location in memory, evaluating a form, editing a function, setting the current frame, and choosing a proceed or restart option. The mouse documentation line at the bottom of the screen tells you what actions are available for the currently highlighted output item.

Besides performing certain mouse operations by clicking directly on displayed Debugger output, you can use menus to perform the usual large variety of other types of operations on Debugger output, just as you can with other kinds of output generated in the Genera software environment.

For more information on using the mouse in the Debugger: See the section "Using the Mouse in the Debugger", page 27.

### 1.4 Overview of Debugger Help Facilities

The Debugger provides online help for Debugger commands and their components, such as keywords. You can get help for all Debugger commands by typing `c-HELP`, which displays brief command descriptions and available key-binding accelerators. For more information about Debugger help: See the section "Getting Help for Debugger Commands", page 19.

## 2. Entering and Exiting the Debugger

Virtually anywhere in Genera, the Debugger is invoked during the signalling of an error to which no condition handlers are bound. The Debugger is invoked not only when errors occur during program execution, but also when errors occur in relation to functions that control various system operations, such as loading patches and executing commands in the Dynamic Lisp Listener.

The Debugger is invoked within the process that signalled the error. Since the Debugger is not a separate process, several distinct processes can all be in the Debugger at the same time, independently.

Usually, entry to the Debugger is triggered by an error. However, you can also enter the Debugger explicitly at any time. You exit the Debugger via the `ABORT` key, the `:Abort` command, or by invoking a proceed or restart handler.

This chapter describes various ways to enter and exit the Debugger.

### 2.1 Entering the Debugger

Enter the Debugger in one of three ways:

- Automatically, by causing an error.
- Explicitly, by pressing `m-SUSPEND` or `c-m-SUSPEND`.
- Through your program execution, by inserting and calling the `break` function or the `zl:dbg` function.

#### 2.1.1 Entering the Debugger by Causing an Error

The Debugger is invoked automatically when errors occur during your program execution, or during the execution of system functions, or when you explicitly cause an error.

##### 2.1.1.1 Error Display

Upon entering the Debugger via an error, you receive an error message and a choice of actions to take. Errors are signalled by the microcode and by Lisp programs by `error` or related functions.

For example, suppose you trigger an error by using an unbound variable, `foo`. The Debugger error display might look like this:

**Trap: The variable FOO is unbound.**

**SI:\*EVAL:**

```

    Arg 0 (SYS:FORM): FOO
    Arg 1 (SYS:ENV): NIL
    --Defaulted args:--
    Arg 2 (SI:HOOK): NIL
s-A, RESUME:    Supply a value to use this time as the value of FOO
s-B, s-sh-C:   Supply a value to store permanently as the value of FOO
s-C:           Retry the SYMEVAL instruction
s-D, ABORT:    Return to Lisp Top Level in Dynamic Lisp Listener 1
→
```

The word *Trap*, *Error*, or *Break* followed by a boldface message, such as the line at the top of this display, indicates you have entered the Debugger. *Trap*, *Error*, and *Break* are the most common causes, although there are others. *Trap*, *Error*, and *Break* have the following meanings:

- *Trap* indicates an error signalled by the microcode.
- *Error* indicates an error signalled by a program.
- *Break* indicates entry to the Debugger by keystroke (*m-SUSPEND* or *c-m-SUSPEND*), the **break** function, or the **zl:dbg** function.

The message that follows describes the error in English – in this example, an unbound variable. The next five lines in the example show the stack frame in which the error occurred, the function that was being called, and the current values of arguments. The next six lines are available proceed and restart options, which are discussed in the next section.

The right-facing arrow at the end of the display (→) is the Debugger's command prompt, which waits for you to enter a command. Multiple arrow prompts indicate recursive invocations of the Debugger. For more information on recursive Debugger invocations: See the section "Using Recursive Debugger Invocations", page 25.

### 2.1.1.2 Debugger Proceed and Restart Options

Whenever you enter the Debugger, either for the first time or recursively, it displays a list of possible actions for you to take. These actions, called *proceed* and *restart options*, allow you to proceed (continue program execution) from the error, leave the Debugger, restart (return to) a previous activity, or take some other action.

A list of proceed and restart options might look like this:

s-A, RESUME:	Supply a value to use this time as the value of F00
s-B, s-sh-C:	Supply a value to store permanently as the value of F00
s-C:	Retry the SYMEVAL instruction
s-D, ABORT:	Debugger command level 1
s-E:	Return to Lisp Top Level in Dynamic Lisp Listener 1

You can select one of these options by pressing the keys that appear in the left-hand column or by clicking on an option with the mouse. All of these options are bound to the SUPER key.

For more information on proceed and restart options: See the section "Proceeding and Restarting in the Debugger", page 20.

### 2.1.2 Entering the Debugger With `m-SUSPEND`, `c-m-SUSPEND`

When you want to enter the Debugger explicitly, without waiting for an error to occur, you can do so in one of two ways:

Press `m-SUSPEND`

Press `c-m-SUSPEND`

If the program you are running is waiting for keyboard input, use `m-SUSPEND`.

If you want to enter the Debugger while your program is actually running, use `c-m-SUSPEND`, which calls the Debugger immediately, at any time, regardless of your program's state.

#### 2.1.2.1 Entering a Break Loop With `SUSPEND`, `c-SUSPEND`

Using `SUSPEND` or `c-SUSPEND`, without the META key modifier, causes entry to a *break loop*. A break loop, also called a *breakpoint loop*, is a Dynamic Lisp Listener read-eval-print loop that comes up on your screen in a special small "breakpoint" window whenever you temporarily suspend an activity, such as Zmacs or Zmail. This allows you to suspend into a Dynamic Lisp Listener instead of typing `SELECT-L` to actually change activities.

Do not confuse this break loop with a Debugger breakpoint. A break loop is a Dynamic Lisp Listener read-eval-print loop, which is activated when you suspend your current activity. A Debugger breakpoint, which you set via the Set Breakpoint command, the `break` function, the `zl:dbg` function, `m-SUSPEND`, or `c-m-SUSPEND`, suspends into the Debugger, usually for the purpose of debugging a program. Once in the Debugger, you can evaluate forms using the Debugger's read-eval-print loop (evaluation environment).

When you want to enter a break loop, you can do so in one of two ways:

Press SUSPEND

Press c-SUSPEND

If the program you are running is waiting for keyboard input, use SUSPEND.

If you want to enter a break loop while your program is actually running, use c-SUSPEND, which brings up the break loop immediately, at any time, regardless of your program's state.

To leave the break loop and return to your previous activity, press the RESUME key.

### 2.1.3 Entering the Debugger With break And zl:dbg Functions

A third way of entering the Debugger is by inserting the **break** or the **zl:dbg** function into your program's source code. These functions can help you detect errors when you place one of them at strategic points in your program – places where you can examine the stack and pinpoint probable causes of errors.

The following paragraphs provide more information on the **break** and the **zl:dbg** functions.

**break** &optional *format-string* &rest *format-args* *Function*

The **break** function is similar to **zl:dbg**. Both functions, when evaluated, cause entry to the Debugger (a Debugger Break). However, **break** takes a *format-string* and *format-args* instead of a process.

The *format-string* is a user-written error message that is printed in the Debugger's Break message whenever **break** is encountered and you enter the Debugger. *format-args* are the FORMAT-style arguments to FORMAT directives in *format-string*.

**break** is a temporary way to insert Debugger breakpoints into your program while you are debugging it. It is not designed for permanent use in your program as a way of signalling errors. Therefore, you would use this function only for the duration of your debugging session. Continuing from **break** will not trigger any unusual recovery action.

**zl:dbg** &optional *process* *Function*

Forces *process* into the Debugger so that you can look at its current state. **zl:dbg** sets up a restart handler for ABORT and RESUME that exits from the **zl:dbg** function back to the original process. The message for this restart handler is "Allow process to continue". You can use :Throw, :Return, :Reinvoke, and other similar Debugger commands when you enter the Debugger via **zl:dbg**.

- With no argument, it enters the Debugger as if an error had occurred for the current process. It is not an error; in particular, **catch-error**

does not handle it. You can include this form in program source code as a means of entering the Debugger. This is useful for breakpoints and causes a special compiler warning.

- With an argument of `t`, rather than a process, window, or stack group, it finds a process that has sent an error notification.

Suppose you are running in process `x` and you use `z!dbg` on some process `y`. Process `y` is forced into the Debugger, no matter what it is doing. Technically, it is "interrupted", similar to how `c-SUSPEND` and `c-m-SUSPEND` work. Process `y` starts running the Debugger, using the stream `*debug-io*`, which gets the same stream as was bound to `*terminal-io*` in process `x`. At this time, process `x` waits in a state called `DBG` until process `y` leaves the Debugger, and so process `x` does not contend for the stream.

## 2.2 Exiting the Debugger

To exit the Debugger, use the `ABORT` key, the `:Abort` command, or invoke a restart option. `ABORT`, which is a very powerful command, takes you out of the process that received the error.

If an error brings you into the Debugger, and you don't want to use the Debugger, you can get back to the top command level in which your program is running by simply pressing `ABORT`. In this case, the top command level is the level in which you were working prior to the Debugger call – the first and only invocation of the Debugger.

If you have made a number of errors, or if you have called the Debugger explicitly several times, then you probably are in the middle of a series of recursive Debugger invocations. In this case, `ABORT` returns you to the previous invocation. If you keep pressing `ABORT`, the invocations unwind until you actually leave the Debugger and return to top level.

If you find yourself in the middle of many recursive Debugger invocations, or if you are in the Debugger's evaluation environment, and you want to leave the Debugger immediately: Press `m-ABORT`, which brings you back to top level immediately.



### 3. Using the Debugger

This chapter offers some general instructions for using the Debugger. Specifically, it covers the following topics:

- Entering a Debugger command
- Getting help for Debugger commands
- Proceeding and restarting in the Debugger
- Evaluating a form in the Debugger
- Using recursive Debugger invocations
- Using the mouse in the Debugger

Descriptions of Debugger commands appear at the end of this chapter.

#### 3.1 Entering a Debugger Command

Entering a Debugger command is almost identical to entering a command in the Command Processor (CP) to a Dynamic Lisp Listener. In fact, you can enter many Debugger commands in both the Debugger and the CP because these commands share the same command table. If you have not done so already, read the information in Book 1 on entering commands in the CP. Specifically: See the section "Communicating with Genera" in *User's Guide to Symbolics Computers*.

When an error brings you into the Debugger, or when you enter the Debugger through `m-SUSPEND`, `c-m-SUSPEND`, `break`, or `zl:dbg`, the Debugger prompts you for commands. The Debugger's command prompt is a right arrow (`->`). Recursive Debugger invocations prompt you with two or more arrows. For example, the third Debugger invocation prompts you with `->->->`. See the section "Using Recursive Debugger Invocations", page 25.

At its command prompt, the Debugger expects a full-form command, such as `:Show Backtrace`, or a command accelerator, such as `c-B`. When giving a full-form command in the Debugger, you must precede the command with a colon. For example:

```
:Show Backtrace
```



If you enter anything other than a colon or an accelerator – anything that is not a Debugger command – the Debugger brings you into its evaluation environment, where you can evaluate Lisp expressions. See the section "Evaluating a Form in the Debugger", page 22.

Because they are implemented as CP commands, Debugger commands have positional arguments, keywords, and command *completion*, which allows you to enter a command without typing the whole command name. You can also edit a Debugger command with the input editor. For more information on positional arguments, keywords, and command completion: See the section "Communicating with Genera" in *User's Guide to Symbolics Computers*.

### 3.1.1 Debugger Command Accelerators

Most Debugger commands have key-binding accelerators. You can enter a command's accelerator instead of its full-form command name. Some commands have only one corresponding accelerator. For example, the accelerator `c-m-F` stands for:

`:Show Function`

Other commands, however, have two or more accelerators that correspond to different variations of the command. For example, the `:Previous Frame` command has five accelerators:

`RETURN, c-P, m-P, c-m-P, c-m-U`

In this case, each accelerator corresponds to a command/keyword combination. For example, `m-P` stands for:

`:Previous Frame :Detailed Yes`

and `c-m-P` stands for:

`:Previous Frame :Internal Yes`

Where applicable, accelerators take a numeric argument to complete the command successfully. For example, if you type the `:Show Backtrace` command, you can specify how many stack frames to display with the `:Nframes` keyword – `:Nframes 2, 3, 4`, and so on. However, if you enter the command accelerator, `c-B`, you can specify how many frames to display by giving a numeric argument. For example, `c-9 c-B` would display nine frames. Likewise, `c-1 c-5 c-B` would display 15 frames.

When you press an accelerator, the Debugger displays an italic message that defines what the accelerator stands for. It then executes the command. For example, when you press the `c-B` accelerator, you get this message:

`→ Control-B Show Backtrace :Nframes 10000 :Internal No :Detailed No`

### 3.1.2 Editing a Debugger Command

When you make a mistake while typing a Debugger command or change your mind about entering the command, you have two choices:

Press `ABORT` and begin again.

Edit your input.

The *input editor* allows you to type, display, and edit a Debugger command. With the input editor, you can edit all Debugger command components – command name, positional arguments, and keywords – before entering the command.

For more information on the input editor: See the section "The Input Editor Program Interface" in *Reference Guide to Streams, Files, and I/O*.

The input editor is also used to edit a form in the Debugger's evaluation environment. For more information on the Debugger's evaluation environment: See the section "Evaluating a Form in the Debugger", page 22.

### 3.1.3 Entering a Debugger Command with the Mouse

You can use the mouse to enter a Debugger command. This is accomplished by simply pointing the mouse at a Debugger command previously displayed in the screen output and clicking on that command. See the section "Using the Mouse in the Debugger", page 27.

## 3.2 Getting Help for Debugger Commands

The Debugger offers you online help. When you press the `HELP` key inside the Debugger, the system displays several help options for you to choose. These options include:

- Pressing `c-HELP`, which displays documentation about all Debugger commands. This documentation consists of brief command descriptions and available key-binding accelerators.
- Pressing the `ABORT` key, which takes you out of the Debugger. (You can enter the `:Abort` command or press `c-Z` instead of pressing `ABORT`.)
- Pressing `c-m-W`, which brings you into the Window Debugger. (You can enter the `:Window Debugger` command instead of pressing `c-m-W`.)

The `REFRESH` key, the `:Show Frame` command, or the `:Show Frame` command accelerator `c-L` clears the screen then redisplay the error message for the current stack frame.

You can also ask for help with keywords. If you do not remember what keywords are available for the command you are entering, press the HELP key after you receive the keywords prompt. The Debugger displays a list of keywords for that command. For example:

```
→ :Previous Frame (keywords) HELP
You are being asked to enter a keyword argument
```

```
These are the possible keyword arguments:
:Detailed          Show locals and disassembled code
:Internal          Show internal interpreter frames
:Nframes          Move this many frames
:To Interesting    Move out to an interesting frame
```

### 3.3 Proceeding and Restarting in the Debugger

Upon entering the Debugger, you might not want to use Debugger commands. Instead, you might want, for example, to continue program execution, leave the Debugger, or return to a previous activity. These alternatives are called *proceeding* and *restarting* in the Debugger. Proceeding means to continue execution from the point where the error occurred. Restarting means to return to a prior activity, such as the Dynamic Lisp Listener or Zmail.

Proceeding and restarting are implemented through a displayed list of possible actions for you to take. These actions are called *proceed* and *restart options*.

#### 3.3.1 Using Debugger Proceed and Restart Options

Whenever you enter the Debugger, either for the first time or recursively, the Debugger displays a list of possible actions for you to take. These actions, called *proceed* and *restart options*, allow you to proceed from the error (continue program execution), leave the Debugger, restart (return to) a previous activity, or take some other action.

A list of proceed and restart options might look like this:

```
s-A, RESUME:    Supply a value to use this time as the value of F00
s-B, s-sh-C:    Supply a value to store permanently as the value of F00
s-C:           Retry the SYMEVAL instruction
s-D, ABORT:     Debugger command level 1
s-E:           Return to Lisp Top Level in Dynamic Lisp Listener 1
```

You can select one of these actions by pressing the keys that appear in the left-hand margin or by selecting an option with the mouse. All of these options are bound to the SUPER key.

Proceed and restart options are assigned to internal proceed handlers or restart handlers respectively. A proceed handler allows you to proceed from the error – continuing execution from the point where the error occurred. For example, you can assign a correct value to an unbound variable then continue execution. A restart handler allows you to unwind the stack – the series of calls that led to the error – and return to a previous system level prior to the error. For example, you can return to a previous Debugger invocation, Zmail, or Zmacs, or you can leave the Debugger and return to the *top level* activity, such as the Dynamic Lisp Listener, as shown above.

### 3.3.1.1 Using ABORT And RESUME in the Debugger

Debugger proceed and restart options are listed in order from the most recent handler that was called to the least recent, oldest handler that was called. The RESUME key is always assigned to the innermost proceed handler or the innermost restart handler if there are no proceed handlers. The ABORT key is always assigned to the innermost restart handler. Pressing the ABORT key usually brings you back to the next previous top-level process in which you were working before the error occurred.

In general, therefore, whenever you want to proceed from the error, press RESUME. Whenever you want to restart the previous activity, press ABORT.

The exact way RESUME works depends on the kind of error that happened. For some errors, there is no standard way to proceed, and the RESUME option just tells you so and returns to the Debugger's command level. For the very common "unbound variable" error, it requests that you supply the Lisp object that should be used in place of the (nonexistent) value of the symbol. For unbound-variable or undefined-function errors, you can also just type Lisp forms to set the variable or define the function, and then press RESUME; execution proceeds after the Debugger asks you to confirm that the new value is acceptable.

The ABORT key, of course, is used in general to exit from the Debugger. See the section "Exiting the Debugger", page 15.

### 3.3.1.2 Supplying a Value to Store Permanently

The value you supply with the RESUME proceed option provides a replacement value but does not change the value of the Lisp object permanently. If you want to change the value permanently, use the proceed option `s-sh-C`, which instructs you to supply a value to store permanently. This option is similar to RESUME, except `s-sh-C` actually sets a variable or defines a function and stores the new value so that the error does not happen again.

### 3.3.1.3 Supplying a Missing Package Prefix

The proceed option `c-sh-P` is only available for such errors as an unbound variable or undefined function when there is a variable or function in another package that has the same name. It permits easy recovery when you forget to supply a package prefix.

### 3.4 Evaluating a Form in the Debugger

You can evaluate a form in the Debugger as easily as you can evaluate a form in a Dynamic Lisp Listener. Evaluation in the Debugger is useful because the Debugger evaluates a form in the context of the function that got the error. All bindings that were in effect at the time the error occurred are in effect when your form is evaluated. You can also evaluate a form using the lexical context of the current frame. For example, you can see the values of lexical variables within LET and LOOP operations. Lexical variables are local variables created temporarily; they exist only for the duration of the lexical operation.

To evaluate a form in the Debugger, simply press a key that is not a command – a character other than a colon or command accelerator key. (As you recall, a full-form Debugger command must begin with a colon.) To evaluate a form, you can type, for example, an open parenthesis. The Debugger gives you the following evaluation prompt:

*Eval (program):*

This *Eval (program):* prompt indicates you are evaluating a form using the *lexical, user-program* context of the current frame. This means you can see the values of Lisp objects, including local variables, at the place where your program execution suspends.

The evaluation prompt comes up the moment you type a non-command character. Your character is immediately placed to the right of the prompt. For example, suppose you type an open parenthesis at the Debugger's right-arrow prompt. This is what happens the moment you type the character:

→ *Eval (program):* (

After it evaluates a form, the Debugger prompts again with the right arrow. If, while typing the form, you change your mind and want to get back to the Debugger's right-arrow prompt, press `ABORT`. Deleting all the characters in the form also brings you back to the Debugger prompt.

The Debugger's evaluation environment is actually a read-eval-print loop that uses the context of the function that received the error. Like a Dynamic Lisp Listener read-eval-print loop, the Debugger's evaluation environment maintains the values of `+`, `*`, and related variables.

If a complex error occurs in the evaluation of the Lisp expression, you are brought into a second Debugger looking at the new error, unless you have specified that your program handle that error. The Debugger prompts with two arrows (`→→`) to show that you are inside two Debuggers. You can get back to the first Debugger by pressing the `ABORT` key. However, if the error is not complex, the abort is done automatically and the original error message is reprinted. See the section "Using Recursive Debugger Invocations", page 25.

Various Debugger commands ask for Lisp objects, such as an object to return or the name of a catch-tag. Whenever it requests a Lisp object, it expects you to type in a form; it will evaluate what you type in. This provides greater generality, since there are objects to which you might want to refer that cannot be typed, such as arrays. If the form you type is not complex (not just a constant form), the Debugger shows you the result of the evaluation and asks you if it is what you intended. It expects a Y or N answer. (See the function `zl:y-or-n-p` in *Programming the User Interface, Volume B*.) If you answer negatively it asks you for another form. To exit the command, just press `ABORT`.

Besides the Debugger's lexical, user-program evaluation environment, the Debugger also has a *dynamic* evaluation environment, created specifically for the task of debugging the debugger. Unless you have to redesign or debug the Symbolics Debugger – an extremely unlikely prospect – *do not* use this evaluation environment. It is used exclusively by Symbolics software development personnel. The prompt for the dynamic evaluation environment is:

*Eval (debugger):*

If you accidentally bring up this prompt, you can change the environment and bring up the *Eval (program):* prompt by entering the `:Use Lexical Environment` command or by pressing `c-x I`, which toggles between the two environments.

The current evaluation environment is established by the previous environment you chose. Therefore, once you're in the lexical program environment, you will stay there until you explicitly enter the `:User Dynamic Environment` command or press `c-x I`.

For more information: See the section "Use Lexical Environment Command", page 50. Also: See the section "Use Dynamic Environment Command", page 49.

### 3.4.1 Editing a Form in the Debugger

When you make a mistake while typing a form in the Debugger or change your mind about entering the form, you can do one of two things:

Press `ABORT` and begin again.

Edit your input.

The *input editor* allows you to type, display, and edit a form in the Debugger's evaluation environment. The input editor is also used for input in Debugger commands and a Dynamic Lisp Listener command processor and read-eval-print loop. For information about editing a Debugger command: See the section "Editing a Debugger Command", page 19. For more information on the input editor: See the section "The Input Editor Program Interface" in *Reference Guide to Streams, Files, and I/O*.

### 3.4.2 Rebound Variable Bindings During Evaluation

When the Debugger evaluates a form, the variable bindings at the point of error are in effect with the following exceptions:

- **\*terminal-io\*** is rebound to the stream the Debugger is using. **dbg:old-terminal-io** is bound to the value that **\*terminal-io\*** had at the point of error.
- **\*standard-input\*** and **\*standard-output\*** are rebound to be synonymous with **\*terminal-io\***; their old bindings are saved in **dbg:old-standard-input** and **dbg:old-standard-output**.
- **\*query-io\***, **\*debug-io\***, and **\*error-output\*** are rebound to be synonymous with **\*terminal-io\***; their old bindings are not directly accessible.
- **+** and **\*** are rebound to the Debugger's previous form and previous value. When the Debugger is first entered, **+** is the last form typed, which is typically the one that caused error, and **\*** is the value of the *previous* form. **++**, **+++**, **\*\***, **\*\*\***, **-**, and **zl:/** are treated in an analogous fashion. See the section "The Lisp Top Level" in *User's Guide to Symbolics Computers*. When the Debugger is exited, all of these variables are restored to their original values; the interactions with the Debugger's read-eval-print loop do not affect the interactions with the top-level Lisp read-eval-print loop.
- **sys:rubout-handler** and **zl:read-preserve-delimiters** are rebound to **nil**, in case the error occurred while in the input editor or the reader.
- **evalhook** is rebound to **nil**, turning off the **zl:step** facility if it was in use when the error occurred. See the section "**evalhook**", page 87.
- **dbg:\*bound-handlers\*** and **dbg:\*default-handlers\*** are rebound to **nil**, preventing conditions signalled by the form the Debugger is evaluating from reaching condition handlers in the program being debugged. This prevents you from accidentally being thrown out of the Debugger.
- **\*print-base\***, **zl:ibase**, and **\*package\*** are checked to insure that they contain legal values. If not, they are set to 10, 10, and **si:pkg-user-package** respectively.

Note that the variable bindings are those in effect in the current frame being examined, unless you are not inheriting the lexical environment, in which case the bindings are those in effect at the point of error.

### 3.5 Using Recursive Debugger Invocations

Whenever you cause an error from within the Debugger, or call the Debugger explicitly from within the Debugger, you are brought into another Debugger.

For example, suppose you used an unbound variable in the Dynamic Lisp Listener. The Debugger is invoked. Then suppose, inside this first Debugger, you reference an undefined function. You are brought into a second Debugger. Then suppose you reference a function that contains a `zl:dbg` function. You are brought into a third Debugger.

In the scenario described above, the three Debugger calls are *recursive Debugger invocations*, where the Debugger causes itself to be called. Each Debugger call is known as a *Debugger command level*. The first call is the first level, the second call is the second level, and the third call is the third level. You can simply refer to the first Debugger, second Debugger, and third Debugger.

If you were to get a backtrace at the third Debugger, you would see that each call to the Debugger appears as a separate stack frame. Like other stack frames, you can unwind the stack – usually with the `ABORT` key – and thereby have each Debugger return to the previous Debugger. The term *unwind* means to return the function in the current frame to the function in the previous frame. Remember: In the third Debugger, you have three *active* Debuggers. They have been called but have not yet returned.

The Debugger command prompt lets you know which Debugger you are in at any given time. For example, three right arrows (`→→→`) indicate you are in the third Debugger. Two right arrows (`→→`) indicate you are in the second. One arrow, of course, indicates you are at the first.

Using the same example, suppose, in a Dynamic Lisp Listener, you reference an unbound variable, `foo`:

**Trap: The variable FOO is unbound.**

**SI:\*EVAL:**

Arg 0 (SYS:FORM): FOO

Arg 1 (SI:ENV): NIL

--Defaulted args:--

Arg 2 (SI:HOOK): NIL

s-A, RESUME: Supply a value to use this time as the value of FOO  
s-B, s-sh-C: Supply a value to store permanently as the value of FOO  
s-C: Retry the SYMEVAL instruction  
s-D, ABORT: Return to Lisp Top Level in Dynamic Lisp Listener 1  
s-E: Restart process Dynamic Lisp Listener 1  
→



Then suppose, within the Debugger, you reference an undefined function, **glitch**:

**Trap: The function GLITCH is undefined.**

**SI:\*EVAL:**

Arg 0 (SYS:FORM): (GLITCH)

Arg 1 (SI:ENV): NIL

*--Defaulted args:--*

Arg 2 (SI:HOOK): NIL

Debugger was entered because an error occurred while evaluating a form in the debugger

s-A, RESUME: Supply a value to use this time as the definition of GLITCH

s-B, s-sh-C: Supply a value to store permanently as the definition of GLITCH

s-C: Retry the FSYMEVAL instruction

s-D, ABORT: Debugger command level 1

s-E: Return to Lisp Top Level in Dynamic Lisp Listener 1

s-F: Restart process Dynamic Lisp Listener 1

→→→

In the example shown above, notice the two right arrows, which indicate entry to the second Debugger. Notice also the restart option, **ABORT**, which allows you to return to the first Debugger. Suppose now, within the second Debugger, you reference **zl:dbg**:

**Break:**

**SI:\*EVAL:**

Arg 0 (SYS:FORM): (ZL:DBG)

Arg 1 (SI:ENV): NIL

*--Defaulted args:--*

Arg 2 (SI:HOOK): NIL

s-A, RESUME: Proceed without any special action

s-B, ABORT: Debugger level 2

s-C: Debugger command level 1

s-D, Return to Lisp Top level in Dynamic Lisp Listener 1

s-E: Restart process Dynamic Lisp Listener 1

→→→→

Now notice the three right arrows, which indicate entry to the third Debugger. Notice also the two restart options, **ABORT** and **s-C**, which allow you to return to the second Debugger and the first Debugger respectively.

Pressing the **ABORT** key is the fundamental way of leaving the current Debugger and returning to the previous Debugger level. If you have amassed many Debugger invocations and want to leave the Debugger entirely and return to top level immediately – in this case Dynamic Lisp Listener 1 – press **m-ABORT**, which

keeps unwinding the stack until you reach top level. `m-ABORT` always gets you back to top level.

The following example shows what happens when you keep pressing `ABORT`, beginning at the third Debugger:

```

→→→ Abort Abort
Debugger command level 2
Back to Trap: The function GLITCH is undefined.
→→ Abort Abort
Debugger command level 1
Back to Trap: The variable FOO is unbound.
→ Abort Abort
Return to Lisp Top Level in Dynamic Lisp Listener 1
Back to Lisp Top Level in Dynamic Lisp Listener 1.

```

Command:

### 3.6 Using the Mouse in the Debugger

Like most other screen output generated in the Genera software environment, Debugger output is mouse sensitive. You can perform some useful debugging operations simply by clicking on output produced by Debugger commands. For example, you can perform the following operations:

- Execute a Debugger command by clicking on any command name that is already displayed on the screen as a result of the command's prior use.
- Set the current stack frame by clicking on a frame's function name displayed in backtrace output.
- Evaluate a form by entering the `:Show Source Code` command, pointing the mouse at a code fragment in the source code output, and pressing `m-Mouse-Middle`.
- Set a Debugger breakpoint on a compiled function by entering the `:Show Compiled Code` command, pointing the mouse at a PC (program counter) line in the disassembled code output, and pressing `c-m-Mouse-Left`.
- Set a Debugger breakpoint on a form in the source code by entering the `:Show Source Code` command, pointing the mouse at a code fragment in the source code output, and pressing `c-m-Mouse-Left`.
- Clear a Debugger breakpoint on a compiled function by entering the `:Show`

Compiled Code command, pointing the mouse at a PC line in the disassembled code output, and pressing `c-m-Mouse-Middle`.

- Clear a Debugger breakpoint on a form in the source code by entering the `:Show Source Code` command, pointing the mouse at a code fragment in the source code output, and pressing `c-m-Mouse-Middle`.
- Monitor the access of a variable or other location by pointing the mouse at a locative, structure slot, or instance variable and pressing `c-m-sh-Mouse-Left`. (When a program or process accesses the monitored location, a Debugger trap is signalled.)
- Unmonitor a variable or other location by pointing the mouse at a locative, structure slot, or instance variable and pressing `c-m-sh-Mouse-Middle`. (When you stop monitoring the access of a location, the Debugger trap is no longer signalled.)
- Edit a function in a Zmacs editor window by pointing the mouse at a function's stack frame and pressing `m-Mouse-Left`.
- Activate a proceed or restart option by clicking on one.
- Perform a **describe** function on a Lisp object by pointing the mouse at any object and pressing `Mouse-Middle`.

Suggested mouse operations are listed in the individual descriptions of some Debugger commands later in this chapter. See the section "Debugger Command Descriptions", page 29. Since so much of the Debugger output is mouse sensitive, the documentation lists only the most useful mouse operations. However, you are encouraged to experiment with the mouse while using the Debugger. You most likely will discover some other mouse or mouse/keyboard capabilities that are particularly suited to your personal debugging style.

Of course, you can perform virtually all of the suggested mouse operations listed in the documentation via momentary menus. As with all other screen output in Genera software environment, you can also use menus and submenus to perform a huge variety of system operations on Debugger output. To perform system or Debugger operations via menus, just point the mouse at the desired piece of Debugger output – a form, function, argument, flavor, instance, locative, or whatever – and click `Mouse-Right`.

### 3.7 Debugger Command Descriptions

This section provides descriptions for all Debugger commands. These commands fall into eight categories according to their functions:

- Commands for viewing a stack frame

- Commands for stack motion

- Commands for general information display

- Commands to continue execution

- Trap commands

- Commands for breakpoints and single stepping

- Commands for system transfer

- Miscellaneous commands

Debugger commands are implemented as Command Processor (CP) commands. There are many Debugger commands that share the global command table with CP commands. Therefore, you can enter these commands in the CP as well as the Debugger. They are:

:Clear All Breakpoints

:Clear Breakpoint

:Disable Condition Tracing

:Edit Function

:Enable Condition Tracing

:Monitor Variable

:Set Breakpoint

:Set Stack Size

:Show Breakpoints

:Show Compiled Code

:Show Monitored Locations

:Show Source Code

:Unmonitor Variable

Note, however, that you must precede every command entered in the Debugger with a colon; for example, you must type `:Set Breakpoint` in the Debugger.

In the sections that follow, Debugger commands are presented in alphabetical order within their logical groups. Each command presentation contains a command format line, a brief command description, and lists of positional arguments, keywords, and useful mouse operations, if any. Key-binding accelerators, if any, appear against the right margin on the command format line. If a command has two or more accelerators, then its accelerators are listed separately with corresponding command/keyword definitions.

Command descriptions use the terms *default* and *mentioned default*. A *default* is the result of entering a Debugger command without a keyword and/or positional argument. A default also means the result of entering a positional argument without a modifier. A *mentioned default* is the result of entering a keyword without a keyword modifier, such as Yes or No. In other words, once you type in the keyword, the Debugger *mentions* the consequences of pressing RETURN without a keyword modifier.

### 3.7.1 Debugger Commands for Viewing a Stack Frame

The Debugger provides commands for displaying information about the current stack frame. Information that you can display includes, for example, argument values, local variable values, disassembled code, source code, and **&rest** arguments. These commands, in alphabetical order, are:

```
:Show Arglist  
  
:Show Argument (c-m-A)  
  
:Show Compiled Code (c-X D)  
  
:Show Frame (REFRESH, c-L, m-L)  
  
:Show Function (c-m-F)  
  
:Show Local (c-m-L)  
  
:Show Rest Argument  
  
:Show Source Code (c-X c-D)  
  
:Show Stack  
  
:Show Value (c-m-V)
```

All of these commands operate in the context of the *current stack frame*. The Debugger knows about the current frame at any given time, and it uses the current frame environment to perform operations according to the suspended state of your program. For example, it evaluates forms in the lexical context of the function suspended in the current frame.

Initially, the current stack frame is the one that signalled the error – either the one that got the microcode-detected error or the one that called **ferror**, **error**, or a related function. The current frame can change, depending on which Debugger operations you perform.

When the Debugger is invoked, it shows you the current frame in the following format:

```
FOO:  
  Arg 0 (X): 13  
  Arg 1 (Y): 1
```

The Debugger displays the name of the function in the current frame, then lists the numbers, names, and values of all arguments in the current frame. In the

case shown above, **foo** was called with two arguments, whose numbers are 0 and 1 and whose names in the Lisp source code are **x** and **y**. The current values of **x** and **y** are 13 and 1 respectively. Numbering of arguments begins with 0. Therefore, argument 0 refers to the first argument, argument 1 refers to the second argument, and so on.

### Show Arglist Command

:Show Arglist c-X c-A

Displays the argument list for the function in the current frame. When you enter this command, the Debugger replies:

The argument list for (*function-name*) is (*argument-names*)

The *function-name* is the name of the function in the current frame – the name of the function that appears when the Debugger is invoked. It is also the name of the function that would appear at the top of the stack if you were to perform a backtrace.

### Show Argument Command

:Show Argument *argument* c-m-A

Displays the value of one or all arguments in the current frame. You can also use the Lisp function (**dbg:arg** *number*) where *number* specifies the number of the argument you want to display. Numbering begins with 0. For example, (**dbg:arg** 3) displays the fourth argument. A numeric argument given with this command's accelerator also specifies the number of the argument you want to display; for example, c-m-3 c-m-A displays the fourth argument. To change the value of an argument, **setf** on (**dbg:arg** *number*).

When you ask to see all arguments – the default for this command – the Debugger displays the arguments in the same way it would display them upon entry to the Debugger. It displays the name of the function in the current frame, then lists the numbers, names, and values of all arguments in that function. When you specify an argument number, the Debugger displays only the value of that argument.

When you are using the lexical context of the current frame, you can evaluate an argument by typing in its name (or clicking on its name using the mouse) in the Debugger's evaluation environment.

The :Show Argument command leaves \* set to the value of the argument so you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack so you can change that argument by calling **setf** on the locative.

*argument*            {*number*, All} The *number* is an integer that specifies which argument you want to display in the current frame. All displays all arguments in the current frame. (Default is All.)

### Show Compiled Code Command

:Show Compiled Code *compiled-function-spec from-pc to-pc*            c-X D

Displays the disassembled code for a function. When you enter this command and specify a compiled-function-spec, the Debugger displays this message:

*Disassembled code for (function):*

where *function* is the name of the compiled function for which you want to see disassembled code. Immediately under this message, the Debugger lists the disassembled code instructions for this function. Each instruction – PUSH, CALL, BRANCH, and so on – is listed on its own line, numbered by the PC (program counter). PCs are numbered in octal (base 8), and numbering begins with 0.

#### *compiled-function-spec*

The name of a compiled function for which you want to see disassembled code. (Default is the function in the current frame.)

#### *from-pc*

The number of the PC at which you want to begin seeing disassembled code. (Default displays all disassembled code.)

#### *to-pc*

The number of the PC at which you want to stop seeing disassembled code. (Default displays disassembled code from PC 0, or from the number specified in *from-pc*, to the last PC in the disassembled code.)

### *Suggested mouse operations*

- To use this command with the mouse: Type in the :Show Compiled Code command. When the Debugger asks you for a compiled-function-spec, point the mouse at the name of a compiled function previously displayed in the output of another command, such as :Show Backtrace or :Next Frame, and click Mouse-Left. (You can do this only when your previous command output includes the name of a compiled function.)
- To set a breakpoint: Point the mouse at a PC in the disassembled code and press c-m-Mouse-Left.
- To clear a breakpoint: Point the mouse at a PC in the disassembled code and press c-m-Mouse-Middle.



**Show Frame Command**

:Show Frame *keywords* REFRESH, c-L, m-L

Displays information about the current frame. (Default redisplay the error message for the current frame then lists the name of the function and its arguments in the current frame.)

*keywords* :Clear Window, :Detailed

:Clear Window {Yes, No} Clears the screen and redisplay at the top of the screen the error message for the current frame. The name of the function and its arguments in the current frame are also displayed. (Default is No. Mentioned default is Yes.)

:Detailed {Yes, No} Redisplay the error message for the current frame then displays detailed information, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

*Key-binding accelerators*

REFRESH, c-L :Show Frame :Clear Window Yes

m-L :Show Frame :Clear Window Yes :Detailed Yes

**Show Function Command**

:Show Function c-m-F

Displays the name of the function in the current frame. You can also use the Lisp function (**dbg:fun**). The Show Function command leaves \* set to the value of the function so that you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the function so that you can change it by calling **setf** on the locative.

**Show Local Command**

:Show Local *local-variable* c-m-L

Displays the value of one or all local variables for the function in the current frame. When you enter this command, the names of local variables and their

values are listed in a sequence: Local 0, Local 1, Local 2, and so on. In this list, locals are numbered in decimal (base 10), and numbering begins with 0.

You can also use the Lisp function (**dbg:loc** *number*) where *number* specifies which local variable you want to display. For example, (**dbg:loc** 3) displays the fourth local variable. A numeric argument given with this command's accelerator also specifies which local variable you want to display; for example, `c-m-3 c-m-L` displays the fourth local variable. To change the value of a local variable, use the **setf** function with (**dbg:loc** *number*).

When you are using the lexical context of the current frame, you can evaluate a local variable by typing its name (or clicking on its name using the mouse) in the Debugger's evaluation environment.

The **:Show Local** command leaves **\*** set to the value of the local variable so you can use the read-eval-print loop to examine it. It also leaves **+** set to a locative pointing to the local variable on the stack so you can change that argument by calling **setf** on the locative.

*local-variable*      {*number*, All} The *number* is an integer that specifies which local variable you want to see in the current frame. All displays all local variables in the current frame. (Default is All.)

### Show Rest Argument Command

**:Show Rest Argument**

Displays the **&rest** argument, if there is one, and formats it neatly. **:Show Rest Argument** sets the value of **\***.

### Show Source Code Command

**:Show Source Code** *compiled-function-spec* c-X c-D

Displays the source code for a function. This command works only when your code resides in an editor buffer. The output is mouse sensitive only when the function is compiled with source locators. When you specify a compiled function for which you want to see source code – for example, **myfunction** – the Debugger displays the source code for **myfunction** beneath the following message:

*Source code for MYFUNCTION:*

If **myfunction** were not compiled with source locators, the Debugger would still display the source code, but the output would not be mouse sensitive. The Debugger would display the source code only after giving you this message:

Function MYFUNCTION has no source locators; the code will not be sensitive.

#### *compiled-function-spec*

The name of a compiled function for which you want to see source code. (Default is the function in the current frame.)

#### *Suggested mouse operations*

When a function has been compiled using source locators – mapping source code to PCs via the editor's `c-m-sh-C` command – you can perform the following mouse operations:

- To use this command with the mouse: Type in the `:Show Source Code` command. When the Debugger asks you for a `compiled-function-spec`, point the mouse at the name of a compiled function previously displayed in the output of another command, such as `:Show Backtrace`, and click `Mouse-Left`.
- To set a breakpoint: Point the mouse at a form (a code fragment) in the displayed source code and press `c-m-Mouse-Left`.
- To clear a breakpoint: Point the mouse at a form (a code fragment) in the displayed source code and press `c-m-Mouse-Middle`.
- To evaluate a code fragment: Point the mouse at a form in the displayed source code and press `m-Mouse-Middle`.

### **Show Stack Command**

`:Show Stack`

Displays all of the local-variable and temporary stack slots in the current frame. This command is very similar to `:Show Local`, except that in addition to local-variable slots, `:Show Stack` displays stack slots that do not necessarily correspond to named local variables. Therefore, `:Show Stack` gives you more information than does `:Show Local`. The output for this command is displayed the way `:Show Local` output is displayed; that is, locals and their values are listed in sequence: Local 0, Local 1, Local 2, and so on. In this list, stack slots are numbered in decimal (base 10), and numbering begins with 0.

### **Show Value Command**

`:Show Value value`

`c-m-V`

Displays one or all values being returned from the function that is being returned. If the frame is not in the process of returning values, the Debugger tells you:

No values are being returned now

:Show Value is useful only when you are using a trap on exit or looking at a frame that is about to return. See the section "Set Trap on Exit Command", page 56.

You can also use the Lisp function (**dbg:val** *number*) where *number* specifies which value to display. Numbering begins with 0. For example, (**dbg:val** 3) displays the fourth value. A numeric value used with this command's accelerator also specifies which value to display; for example, `c-n-3 c-n-U` displays the fourth value. To change a particular value being returned from a frame, use **setf** on (**dbg:val** *number*).

The :Show Value command leaves \* set to the value of the argument, so you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack so you can change that argument by calling **setf** on the locative.

*value*                    {*number*} The *number* is an integer that specifies which value to display.

### 3.7.2 Debugger Commands for Stack Motion

The Debugger provides commands that allow you to move up and down the stack. The term *move* in the context of these commands means to make another frame the current frame. For example, moving to the top of the stack makes the most recent frame – the frame where the error occurred – the current frame.

Moving down the stack takes you back in time toward the oldest, least-recent frame. Moving up the stack takes you forward in time toward the newest, most-recent frame, which is usually the call to the Debugger itself.

Stack motion commands not only traverse the stack, but they also display information about the frame to which you move. Most of these commands can optionally display local variables, disassembled code, and internal interpreter frames.

The motion commands, in alphabetical order, are:

:Bottom Of Stack (m->)

:Find Frame (c-S)

:Next Frame (LINE, c-N, m-N, c-m-N)

:Previous Frame (RETURN, c-P, m-P, c-m-P, c-m-U)

:Set Current Frame

:Top Of Stack (m-<)

### Bottom of Stack Command

:Bottom Of Stack *keyword* m->

Moves to the bottom of the stack, displays the least recent frame, and makes that frame current. When you enter this command, the Debugger displays the name of the function at the bottom of the stack, followed by its arguments.

<i>keyword</i>	:Detailed
:Detailed	{Yes, No} Displays detailed information about the frame at the bottom of the stack, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

### Find Frame Command

:Find Frame *string keywords* c-S

Searches the stack for a frame's function name that contains a specified string and makes that frame current. When you enter this command, the Debugger displays the name of the function in the specified frame, followed by its arguments.

<i>string</i>	A <i>string</i> that can be part or all of a function name.
<i>keywords</i>	:Detailed, :Reverse
:Detailed	{Yes, No} Displays detailed information about the specified

frame, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

**:Reverse** {Yes, No} Searches backwards, toward the most recent frame, for the specified frame. (Default is No. Mentioned default is Yes.)

### Next Frame Command

**:Next Frame** *keywords* LINE, c-N, m-N, c-m-N

Moves down one frame, to the next less-recent frame – the calling frame – displays information about that frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the next frame, followed by its arguments. A numeric argument given with this command's accelerators, as well as the :Nframes keyword, specifies how many frames to move down; for example, c-3 c-N moves down three frames.

*keywords* :Detailed, :Internal, :Nframes

**:Detailed** {Yes, No} Displays detailed information about the next frame, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

**:Internal** {Yes, No} Displays internal interpreter frames in the next frame. Ordinarily, when running interpreted code, the Debugger tries to skip over frames that belong to functions of the interpreter, such as **si:eval**, **prog**, and **cond**, and only show "interesting" functions. (Default is No. Mentioned default is Yes.)

**:Nframes** {*number*} Specifies how many frames you want to move down. The *number* signifies that you want to move down to the *n*th frame from the current frame. (Default is 1.)

### Key-binding accelerators

LINE, c-N :Next Frame :Nframes 1

n-N               :Next Frame :Detailed Yes :Nframes 1  
 c-m-N             :Next Frame :Internal Yes :Nframes 1

### Previous Frame Command

:Previous Frame *keywords*                               RETURN, c-P, n-P, c-m-P, c-m-U

Moves up one frame, to the next most-recent frame – the frame that the current frame called – displays information about that frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the previous frame, followed by its arguments. A numeric argument given with this command's accelerators, as well as the :Nframes keyword, specifies how many frames to move up; for example, c-3 c-P moves up three frames.

*keywords*               :Detailed, :Internal, :Nframes, :To Interesting

:Detailed            {Yes, No} Displays detailed information about the previous frame, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

:Internal            {Yes, No} Displays internal interpreter frames in the previous frame. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as **si:eval**, **prog**, and **cond**, and only show "interesting" functions. (Default is No. Mentioned default is Yes.)

:Nframes            {*number*} Specifies how many frames you want to move up. The *number* signifies that you want to move up to the *n*th frame from the current frame. (Default is 1.)

:To Interesting    {Yes, No} Moves to the next previous frame that is interesting (non-interpreter), skipping over interpreter frames. (Default is No. Mentioned default is Yes.)

### Key-binding accelerators

RETURN, c-P       :Previous Frame :Nframes 1  
 n-P               :Previous Frame :Detailed Yes :Nframes 1  
 c-m-P             :Previous Frame :Internal Yes :Nframes 1

c-m-U                   :Previous Frame :To Interesting Yes

### Set Current Frame Command

:Set Current Frame *stack-frame*

Makes the stack frame that you specify with the mouse become the current frame.

*stack-frame*           A *stack frame* that you select with the mouse.

#### *Suggested mouse operations*

- To set the current frame: Display the stack with the :Show Backtrace command, point the mouse at the stack frame you want to make current, and click Mouse-Left.

### Top of Stack Command

:Top Of Stack *keyword* m-<

Moves to the top of the stack – the frame where the error occurred – displays the most recent frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the frame at the top stack, followed by its arguments.

*keyword*                   :Detailed

:Detailed                 {Yes, No} Displays detailed information about the frame at the top of the stack, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

### 3.7.3 Debugger Commands for General Information Display

The Debugger provides commands that allow you to examine the Lisp control stack and display general information about your program's execution as it relates to the error that triggered entry to the Debugger. Information that you display, for example, can be the value of \*, special variable bindings, catch blocks, condition handlers, instructions, standard value warnings, proceed options, and so on.

The most powerful information-display command is :Show Backtrace, which displays the Lisp control stack. The stack keeps a record of all active functions.



The term *active* refers to a function that has been called but has not yet returned. For example, if you call **foo** at Lisp's top level, and it calls **bar**, which in turn calls **baz**, and **baz** gets an error, then a *backtrace* displays this call history. Functions **foo**, **bar**, and **baz** appear on the stack because they have been called but have not yet returned. A backtrace, therefore, traces the execution of program functions and system functions back in time, and the Debugger displays the sequence of calls that led to the error.

The `:Show Backtrace` command can display a brief backtrace with only function names in a call history sequence, or it can display backtraces with more detailed information, such as arguments, local variables, disassembled code, and internal interpreter frames. Using the the **foo/bar/baz** example mentioned above, a brief backtrace of that call history might look like this:

```
BAZ ← BAR ← FOO ← EVAL ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

In the example shown above, the arrows indicate the direction of calling. See the section "Show Backtrace Command", page 45.

The general information display commands, in alphabetical order, are:

:Analyze Frame (c-m-Z)  
:Describe Last (c-m-D)  
:Show Backtrace (c-B, m-B, c-m-B)  
:Show Bindings (c-X B)  
:Show Catch Blocks  
:Show Condition Handlers  
:Show Instruction (c-m-I)  
:Show Lexical Environment  
:Show Proceed Options  
:Show Special  
:Show Standard Value Warnings  
:Symeval In Last Instance (c-X c-I)  
:Use Dynamic Environment (c-X I)  
:Use Lexical Environment (c-X I)

### Analyze Frame Command

:Analyze Frame c-m-Z

Analyzes the erroneous frame and locates the source code of the current error. Whenever your program blows up unexpectedly, for example, due to an incorrect argument value or undefined function, you can use the :Analyze Frame command to walk back up the stack and locate the origin of the error.

Specifically, the :Analyze Frame command can locate the source-code origin of these type of errors:

Incorrect argument values

Invalid or undefined functions

Unclaimed messages

Wrong number of arguments

If :Analyze Frame does not operate on a particular kind of error, the Debugger tells you:

*There is nothing to analyze in this frame.*

:Analyze Frame tells you the name of the function where the error occurred, moves to the previous frame, and examines the code in the previous frame. If it does not find the origin of the error in that frame, it keeps moving up the stack, examining code frame by frame. For each frame, the Debugger displays the name of the "bad argument" that received the error as well as the name of the function that passed the error – the calling function. It also highlights the bad argument and calling function in boldface type and displays the source code.

The last frame the Debugger displays is the frame that caused the error.

Suppose a bad argument, **foo**, was passed to a function, **myfunction** – the place where the error occurred – and **foo** originated from another function, **glitch**. The Debugger would display the source code of **myfunction** beneath the following message:

*Error occurred in MYFUNCTION:*

Then the Debugger would tell you:

*Probably bad argument FOO*

followed by this message:

*Called from GLITCH:*

The Debugger would then display the source code for **glitch**. If the bad argument, **foo**, had not originated from **glitch**, the Debugger would have kept crawling up the stack, and, for each frame, would have displayed the probable bad argument and the source code of the calling function.

Suppose you execute a function, **test**, without arguments, and **test** calls another function, **number-test**, which expects one argument, **n**. Via the :Analyze Frame command, the Debugger would display the following information:

*Bad call occurred in:*

```
(DEFUN TEST ()
  (NUMBER-TEST))
```

*Correct arguments to NUMBER-TEST are (N)*

*Correct arguments to NUMBER-TEST are (N)*

### Describe Last Command

:Describe Last

c-m-D

Executes the Lisp **describe** function on the most recently displayed value and leaves \* set to that value.

#### *Suggested mouse operations*

- To perform a **describe** function on any Lisp object: Point the mouse at any object in the output and click `Mouse-Middle`.

### Show Backtrace Command

:Show Backtrace *keywords*

c-B, m-B, c-m-B

Displays a backtrace of the stack. The default displays a brief backtrace of the stack.

A brief backtrace displays just the names of active function calls in the sequence in which they were called. In the display, each function points to the function it calls. For example:

```
BAZ ← BAR ← FOO ← EVAL ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

If you want a backtrace with more detailed information and/or with internal interpreter frames, use the `:Detailed` and `:Internal` keywords described below. See also the definitions of command accelerators below.

A numeric argument given with this command's accelerators, as well as the `:Nframes` keyword, specifies how many frames to display in the stack; for example, `c-9 c-B` displays nine frames.

*keywords*           :Detailed, :Internal, :Nframes

- :Detailed           {Yes, No} Displays a detailed backtrace of the stack, with arguments and their values. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)
- :Internal           {Yes, No} Displays internal interpreter frames in the backtrace. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as `si:*eval`, `prog`, and `cond`, and only show "interesting" functions. (Default is No. Mentioned default is Yes.)

**:Nframes**            {*number*} Designates how many frames to display in the backtrace. Enter a *number* to specify the number of frames to display. (Default is 10000.)

### *Key-binding accelerators*

**c-B**                    :Show Backtrace :Nframes 10000 (brief backtrace)  
**n-B**                    :Show Backtrace :Detailed Yes :Nframes 10000  
**c-n-B**                 :Show Backtrace :Internal Yes :Nframes 10000

### **Show Bindings Command**

**:Show Bindings** *keywords* c-X B

Displays the special variable bindings for one or more frames. When you enter this command, the Debugger displays special variable bindings beneath this message:

*Names and values of specials bound in this frame:*

**keywords**            :All, :Matching

**:All**                 {Yes, No} Displays bindings for all frames in the stack. (Default is No. Mentioned default is Yes.)

**:Matching**         {*string*} Displays only the bindings for special variables whose symbol names contain a *string* that you specify. (Default is the current frame.)

### **Show Catch Blocks Command**

**:Show Catch Blocks** *keyword*

Displays the active catch blocks for the current frame or for all frames. When you enter this command, the Debugger displays information in this format:

*Open catch blocks and unwind-protects in this frame:*  
 Throwing to tag *tag-name* returns to *frame* at *location*  
 with value(s)

The *tag-name* is the name of the symbol that is catching the form. The *frame* is the name of the frame's function to which a **throw** operation returns. The *location* is a PC (program counter) line number in disassembled code.

*keyword* :All  
 :All {Yes, No} Displays active catch blocks for all frames in the stack. (Default is No. Mentioned default is Yes.)

### Show Condition Handlers Command

:Show Condition Handlers *keyword*

Displays the condition handlers for the current frame or for all frames. Here is an example of what the Debugger displays when you enter this command for the current frame:

```
→ :Show Condition Handlers
Bound Handlers established in this frame:
CONDITION-CASE handler for SYS:PARSE-ERROR
→
```

If the frame shown in the example above were not the current frame, and you used the :All keyword, the Debugger would display the name of the frame along with the condition handler information. For example:

```
→ :Show Condition Handlers (keywords) :All
For frame (DEFUN-IN-FLAVOR SI:INPUT-EDITOR-READ ... ):
Bound Handlers established in this frame:
CONDITION-CASE handler for SYS:PARSE-ERROR
→
```

*keyword* :All  
 :All {Yes, No} Displays condition handlers for all frames in the stack. (Default is No. Mentioned default is Yes.)

### Show Instruction Command

:Show Instruction

c-m-I

Displays the instruction that was just trapped in the Debugger or the instruction that would be executed next if you were to perform a single step operation. Here is an example of what the Debugger displays when you enter this command:

```
→ :Show Instruction
In (FLAVOR:METHOD :INPUT-EDITOR SI:INTERACTIVE-STREAM) at PC 160:
PUSH-NIL
→
```

## Show Lexical Environment Command

:Show Lexical Environment

Displays the lexical (local program) environment of the current frame, as established by the lexical ancestors of the frame. When you enter this command, the Debugger displays lexical (local) variables beneath this message:

*Lexically inherited variables:*

If the current frame has no lexical environment, the Debugger tells you:

This frame was not lexically called.

## Show Proceed Options Command

:Show Proceed Options

Displays all of the currently available proceed and restart options. Here is an example of what the Debugger displays when you enter this command:

```
→ :Show Proceed Options
s-A, RESUME:    Supply a value to use this time as the value of F00
s-B, s-sh-C:   Supply a value to store permanently as the value of F00
s-C:           Retry the SYMEVAL instruction
s-D, ABORT:    Return to Lisp Top Level in Dynamic Lisp Listener 1
→
```

### *Suggested mouse operations*

- To activate a proceed handler with the mouse: Display the proceed options with the :Show Proceed Options command, point the mouse at a proceed option, and click Mouse-Left.

## Show Special Command

:Show Special *symbol keyword*

Displays the special variable binding of a symbol in the context of the current frame's environment.

*symbol*            A symbol whose special variable binding you want to see.

*keyword*            :Environment

:Environment    {Program, Debugger, Streams} Evaluates and displays the symbol in the environment that you specify. Program specifies a program you are debugging. Debugger and Streams specify

that you are debugging the Debugger. (Default is the environment of the current frame. Mentioned default is Program.)

### Show Standard Value Warnings Command

:Show Standard Value Warnings

Displays more detailed information about standard variables that have been re-bound. Here is an example of what the Debugger displays when you enter this command:

```
→ :Show Standard Value Warnings
The following standard values were bound:
  Rebinding CP:*COMMAND-TABLE* to #<COMMAND-TABLE User #o260252757>
  (old value was #<COMMAND-TABLE Debugger #o261747137>)
→
```

If no standard variables have been re-bound, the Debugger tells you:

```
There were no standard values which required binding
```

### Symeval in Last Instance Command

:Symeval In Last Instance *symbol* c-X c-I

Evaluates a symbol as an instance variable in the context of the last instance to have been typed out.

*symbol*            A symbol to be evaluated.

### Use Dynamic Environment Command

:Use Dynamic Environment c-X I

Changes the current evaluation mode from the lexical (local program) environment to the dynamic (global debugger) environment. Unless you debug your own Debugger, do not use this command. The :Use Dynamic Environment command is used by Symbolics development personnel who debug the Debugger. If you have entered the dynamic evaluation environment accidentally, you can get back to the lexical evaluation environment by entering the :Use Lexical Environment command or by pressing c-X I, which toggles between the two evaluation environments. The dynamic evaluation prompt is:

*Eval (debugger):*



## Use Lexical Environment Command

:Use Lexical Environment c-X I

Changes the current evaluation mode from the dynamic (global debugger) environment to the lexical (local program) environment. When the Debugger is in this evaluation environment, you can examine local variables and arguments by simply typing their names, and you can use internal functions by name – functions defined with **flet** or **labels**. See the section "Evaluating a Form in the Debugger", page 22. The lexical evaluation prompt is:

*Eval (program):*

The c-X I accelerator toggles between the lexical evaluation environment and the dynamic evaluation environment.

### 3.7.4 Debugger Commands to Continue Execution

The Debugger provides commands that continue or restart execution. These commands, in alphabetical order, are:

:Abort (ABORT, c-Z)

:Disable Aborts

:Enable Aborts

:Proceed (RESUME)

:Reinvoke (c-n-R)

:Return (c-R)

:Throw (c-T)

### Abort Command

:Abort ABORT, c-Z

Depending on the context of its use: Returns to either top level or the previously invoked Debugger. Executes the abort instruction that appears in the list of proceed and restart options. :Abort is used to exit the Debugger. See the section "Exiting the Debugger", page 15. You can use the ABORT key in place of this command.

**Disable Aborts Command****:Disable Aborts**

Disables the use of the **:Abort** command. **:Disable Aborts** is useful for making sure you do not abort something accidentally.

**Enable Aborts Command****:Enable Aborts**

Enables the use of the **:Abort** command.

**Proceed Command****:Proceed**

RESUME

Depending on the context of its use: Continues the execution of the program or process that has been suspended, executes the proceed-handler instruction that appears in the list of proceed and restart options, or returns to the previously invoked Debugger. You can use the RESUME key in place of this command.

**Reinvoke Command****:Reinvoke *keyword***

c-m-R

Restarts execution of the function in the current frame. Any numeric argument given with this command's accelerator, as well as the **:New Args** keyword, prompts you for new argument values. If the function has been redefined – perhaps you edited the function to fix a bug – the new definition is used. The **:Reinvoke** command asks for confirmation before restarting the frame.

*keyword***:New Args**

**:New Args** {Yes, No} Prompts you to supply new argument values for the reinvoked frame. (Default is No, which reinvokes the frame using current argument values. Mentioned default is Yes.)

**Return Command****:Return**

c-R

Returns from the current frame. This command prompts for as many values as the caller needs. You must enter values acceptable to the current frame's caller. For each value, the Debugger prompts you for a form, which it evaluates. It returns the resulting values, possibly after confirming them with you. If no

values are expected, it requests confirmation before returning. The `:Return` command is useful when you want to simulate the return of a frame's execution, which was halted for some reason.

### Throw Command

`:Throw symbol form` c-T

Executes a Lisp `throw` function and throws the result of evaluating *form* to the tag named by *symbol*. You can also use the Lisp function `throw`.

*symbol*            A catch tag.

*form*             A *form* to evaluate. The returned values from this evaluation are thrown to *symbol*.

### 3.7.5 Debugger Trap Commands

The Debugger provides commands associated with Debugger traps. These commands, in alphabetical order, are:

`:Clear Trap On Call (c-X c-C)`

`:Clear Trap On Exit (c-X c-E)`

`:Disable Condition Tracing (c-X T)`

`:Enable Condition Tracing (c-X T)`

`:Monitor Variable`

`:Proceed Trap On Call (c-X m-C)`

`:Restart Trap On Call (c-X c-m-C)`

`:Set Trap On Call (c-X C)`

`:Set Trap On Exit (c-X E)`

`:Show Monitored Locations`

`:Unmonitor Variable`

A *trap* suspends a function's execution and, if there is no condition handler, causes entry to the Debugger. For example, a trap might be signalled when your program executes an illegal instruction, such as division by 0. Unless your program is prepared to handle the trap, the Debugger is entered.

The `:Monitor Variable` command also causes a trap and Debugger entry. This command triggers a *monitor trap* whenever a process accesses a special variable. If you have many different processes accessing a special variable, and you want to identify them all, you can simply specify the variable to be monitored. The trap occurs when that variable is referenced. You can also monitor instance variables and structure slots by clicking on them with the mouse. `:Monitor Variable` is useful if you want to keep track of and debug the interactions between the accessing processes. See the section "Monitor Variable Command", page 54.

The `:Enable Condition Tracing` command also signals a trap when you suspect a condition handler is broken and want to debug that handler. If you receive recursive error messages due to a defective handler, use `:Enable Condition Tracing` to cause a trap and enter the Debugger before the condition is signalled. See the section "Enable Condition Tracing Command", page 54.

Once in the Debugger, you can explicitly set traps by using the `:Set Trap On Call` and `:Set Trap On Exit` commands. A trap on exit suspends execution outside the called function, immediately after the function has returned. A trap on call suspends execution inside the called function, immediately before the first instruction.

The `RESUME` key can be used to continue returning or throwing whenever execution is suspended in a trap. When a trap on exit is set for a frame, throwing through that frame still signals the trap.

The `ABORT` key can be used to bypass the trap on exit.

The `:Set Trap On Call`, `:Proceed Trap On Call`, and `:Restart Trap On Call` commands have the following restriction: If you are metering all functions in a particular process, you cannot use trap on call in that process while metering is enabled.

### Clear Trap on Call Command

`:Clear Trap On Call` c-X c-C

Clears trap on call for the current frame.

### Clear Trap on Exit Command

`:Clear Trap On Exit keyword` c-X c-E

Clears trap on exit for the current frame or for all frames. Any numeric argument given with this command's accelerator clears trap on exit for all frames.

*keyword*           :All

:All                {Yes, No} Clears traps on exit for all frames in the stack.  
(Default is No. Mentioned default is Yes.)

### Disable Condition Tracing Command

:Disable Condition Tracing c-x T

Disables condition tracing. The c-x T accelerator toggles between :Disable Condition Tracing and :Enable Condition Tracing.

### Enable Condition Tracing Command

:Enable Condition Tracing *condition keyword* c-x T

Enables condition tracing. That is, this command allows you to debug an error handler when it does not work properly. For example, when you receive continuous, recursive error messages due to a defective error handler, you can use :Enable Condition Tracing to cause a trap and enter the Debugger before the condition is signalled. Once in the Debugger, you can debug and fix the handler.

You should use this command only if you code your own error handlers. If you do not code your own handlers, and suspect there is a bug in a handler, send a bug report to your Symbolics customer representative.

Any numeric argument given with this command's accelerator sets **trace-conditions** to **t**. The c-x T accelerator toggles between :Enable Condition Tracing and :Disable Condition Tracing.

*condition* {t, nil, *condition*} **t** enters the Debugger when any condition is signalled. **nil** turns off condition tracing previously specified by **t**. *condition* is a condition flavor, which causes entry to the Debugger when any flavor built on *condition* is signalled.

*keyword* :Conditional

:Conditional {Always, Mode-Lock, Never, Once} Enables condition tracing according to certain conditions. **Always**: enables condition tracing in all cases. **Mode-Lock**: enables condition tracing only when the MODE LOCK key is depressed. **Never**: has the effect of disabling condition tracing. **Once**: enables condition tracing only for the first time a condition is raised. (Default is Always.)

### Monitor Variable Command

:Monitor Variable *symbol keywords*

Monitors the access of a special variable. This command arranges for a trap to be signalled when any process accesses the monitored location. This command is used to answer the question: "What program or process is reading or writing this location in memory?". This is particularly useful when there are several processes

sharing some data structures, and you want to debug the interactions between the processes.

*symbol*            The name of a symbol whose location in memory you want to monitor. Enter the name of a symbol and, optionally, its Value-Cell or Function-Cell. (See the :Cell keyword description below.)

*keywords*        :Boundp, :Cell, :Locf, :Makunbound, :Read, :Write

:Boundp            {Yes, No} Monitors the location for **boundp** operations. (Default is No.)

:Cell                {Value-Cell, Function-Cell} Specifies the cell that you want to monitor within the *location*. The Debugger gives you two choices: Value-Cell or Function-Cell. (Default is Value-Cell.)

:Locf                {Yes, No} Monitors the location for **locf** operations. (Default is No.)

:Makunbound        {Yes, No} Monitors the location for **makunbound** operations. (Default is No.)

:Read                {Yes, No} Monitors the location for reads. (Default is No.)

:Write                {Yes, No} Monitors the location for writes. (Default is Yes.)

#### *Suggested mouse operations*

- To monitor a location: Point the mouse at a locative, structure slot, or instance variable and press `c-m-sh-Mouse-Left`.
- To unmonitor a location: Point the mouse at a locative, structure slot, or instance variable that was previously monitored and press `c-m-sh-Mouse-Middle`.

#### **Proceed Trap on Call Command**

:Proceed Trap On Call

`c-X m-C`

Resumes execution of the function in the current frame after setting trap on call. Use this command when you want to suspend execution at the entry to the next called function immediately. The :Restart Trap On Call command is similar, except that it restarts execution from the beginning of the current function before it suspends execution at the next called function. See the section "Restart Trap on Call Command", page 56. Using the :Proceed Trap On Call command is identical to using the :Set Trap On Call and :Proceed commands successively. The trap on call suspends execution inside the called function, immediately before the first instruction. See the section "Set Trap on Call Command", page 56.

*Note:* The Debugger might mistake this command for the `:Proceed` command if you attempt to type in the full command name. To avoid this problem, use the `c-X m-C` accelerator, surround the command name in quotes (excluding the colon), or type in:

```
:p t COMPLETE
```

to complete the command properly.

### Restart Trap on Call Command

```
:Restart Trap On Call c-X c-m-C
```

Restarts execution of the function in the current frame, but first sets trap on call. Use this command when you want to restart execution of the current frame then immediately suspend execution at the entry to the next called function. The `:Proceed Trap On Call` command is similar, except that it resumes execution from wherever execution is suspended within the function instead of restarting execution from the beginning of the function. See the section "Proceed Trap on Call Command", page 55. Using the `:Restart Trap On Call` command is identical to using the `:Set Trap On Call` and `:Reinvoke` commands successively. The trap on call suspends execution inside the called function, immediately before the first instruction. See the section "Set Trap on Call Command", page 56.

### Set Trap on Call Command

```
:Set Trap On Call c-X C
```

Sets trap on call for the next function called in the current frame. Use this command when you want to suspend execution at the entry of the next called function. (This command also sets trap on exit for the next called function.) The trap occurs only for the first time your program execution encounters the called function. A trap on call suspends execution inside the called function, immediately before the first instruction.

### Set Trap on Exit Command

```
:Set Trap On Exit keyword c-X E
```

Sets trap on exit for the current frame or for all frames. The trap on exit occurs only for the first time your program execution returns the called function. Any numeric argument given with this command's accelerator sets traps on exit for all frames. When a trap on exit is set for a frame, throwing through that frame, via a Lisp `throw` function, still signals the trap.

*keyword*           :All  
           :All           {Yes, No} Sets traps on exit for all frames in the stack.  
                           (Default is No. Mentioned default is Yes.)

### Show Monitored Locations Command

:Show Monitored Locations

Displays all variables and other locations in memory that you are monitoring via the :Monitor Variable command, the **dbg:monitor-location** function, and so on.

### Unmonitor Variable Command

:Unmonitor Variable *symbol keyword*

Stops monitoring one or all special variables in memory.

*symbol*            {*location*, RETURN} A *location* specifies one location that you want to stop monitoring. Enter the name of a symbol and, optionally, its Value-Cell or Function-Cell. (See the :Cell keyword description below.) Press the RETURN key if you want to stop monitoring all locations.

*keyword*           :Cell

:Cell            {Value-Cell, Function-Cell} Specifies which cell within the location you want to stop monitoring. The Debugger gives you two choices: Value-Cell or Function-Cell. (Default is Value-Cell.)

### *Suggested mouse operations*

- To unmonitor a location: Point the mouse at a locative, structure slot, or instance variable that was previously monitored and press `c-m-sh-Mouse-Middle`.

## 3.7.6 Debugger Commands for Breakpoints and Single Stepping

The Debugger provides breakpoint and single-step commands.

Like a trap, a Debugger *breakpoint* is also a suspension of a function's execution. Unlike a trap on call or trap on exit, any breakpoint that you set suspends execution every time your program encounters the breakpoint. You can set a breakpoint with the :Set Breakpoint command as well as other ways, listed below. Breakpoints are useful for examining data at strategic points in your program while your execution is frozen.



When you enter the Debugger via breakpoint, the Debugger displays the word **Break** in the top line of the error display. A Debugger breakpoint can be signalled by:

- Using the `:Set Breakpoint` command. See the section "Set Breakpoint Command", page 60.
- Performing mouse operations on the code fragments and disassembled code instructions output by the `:Show Source Code` and `:Show Compiled Code` commands respectively. See the section "Show Source Code Command", page 35. Also: See the section "Show Compiled Code Command", page 33.
- Pressing `m-SUSPEND` or `c-m-SUSPEND`. See the section "Entering the Debugger With `m-SUSPEND`, `c-m-SUSPEND`", page 13.
- Inserting the `break` or `zl:dbg` function into your program's source code. See the section "Entering the Debugger With `break` And `zl:dbg` Functions", page 14.

Do not confuse a Debugger breakpoint with a *break loop*. A break loop is a Dynamic Lisp Listener read-eval-print loop, which is activated when you suspend your current activity, via `SUSPEND` or `c-SUSPEND`. A Debugger breakpoint suspends into the Debugger, usually for the purpose of debugging a program.

You should set breakpoints only in your program's source code. Do not set a breakpoint in a system function – any code that the system depends on for its operations. Placing a breakpoint in a system function can produce dangerous results because your breakpoint may be encountered by other system functions. A breakpoint in the following types of functions can be particularly dangerous:

Input/Output functions

Input Editor functions

Storage system functions

Hardware I/O functions

Garbage collecting functions

The term *single stepping* refers to the process of executing instructions, one instruction at a time. That is, the `:Single Step` command executes the next instruction, then suspends execution. The pattern becomes execute-suspend, execute-suspend, execute-suspend, and so on. The `:Single Step` command only operates on compiled code. To single step through interpreted code, use the `Step` facility or the `:step` option in the Trace facility. See the section "Stepping

Through an Evaluation", page 85. Also: See the section "Tracing Function Execution", page 73. The :Single Step command steps over compiled functions. To step into a compiled function, use the :Set Trap On Call command on the function in which you want to step, then use the :Single Step command.

Commands for breakpoints and single stepping, in alphabetical order, are:

:Clear All Breakpoints

:Clear Breakpoint

:Set Breakpoint

:Show Breakpoints

:Single Step (c-sh-S)

### Clear All Breakpoints Command

:Clear All Breakpoints *compiled-function-spec*

Clears all breakpoints in the current frame's function or in any other compiled function.

*compiled-function-spec*

The name of a compiled function in which you want to clear breakpoints. (Default clears all breakpoints in the current frame's function.)

### Clear Breakpoint Command

:Clear Breakpoint *compiled-function pc*

Clears a breakpoint.

*compiled-function* The name of a *compiled function* in which you want to clear a breakpoint.

*pc* The PC (program counter) at which you want to clear a breakpoint.

### *Suggested mouse operations*

- To clear a breakpoint in a compiled function: Display disassembled code with the :Show Compiled Code command, point the mouse at a PC, and press c-n-Mouse-Middle.

- To clear a breakpoint in a code fragment: Display the code with the :Show Source Code command, point the mouse at a code fragment, and press `c-n-Mouse-Middle`.

### Set Breakpoint Command

:Set Breakpoint *compiled-function pc*

Sets a breakpoint.

*compiled-function* The name of a *compiled-function* in which you want to set a breakpoint.

*pc* The PC (program counter) at which you want to set a breakpoint.

*keywords* :Action, :Conditional

- :Action {Show-All, Show-Args, Show-Locals, *expression*} Specifies an action to take when the breakpoint is encountered. Show-All: Displays arguments and local variables. Show-Args: Displays arguments and no local variables. Show-Locals: Displays only local variables. Give an *expression* if you want it to be evaluated in the lexical context of the frame. (Default is no action. Mentioned default is Show-All.)
- :Conditional {Always, Mode-Lock, Never, Once, *expression*} Executes the breakpoint trap according to certain conditions. Always: The breakpoint is always taken. Mode-Lock: The breakpoint is taken only when the MODE LOCK key is depressed. Never: The breakpoint is never taken. Once: The breakpoint is taken only for the first time it is encountered. Give an *expression* if you want it to be evaluated in the lexical context of the frame. (Default is Always.)

### Suggested mouse operations

- To set a breakpoint in a compiled function: Display disassembled code with the :Show Compiled Code command, point the mouse at a PC, and press `c-n-Mouse-Left`.
- To set a breakpoint in a code fragment: Display the code with the :Show Source Code command, point the mouse at a code fragment, and press `c-n-Mouse-Left`.

### Show Breakpoints Command

:Show Breakpoints

Displays all of the currently set breakpoints.

### Single Step Command

:Single Step

c-sh-S

Executes one instruction at a time and steps over function calls. This command works only on compiled code. For interpreted code, use the Step facility or the :step option in the Trace facility. For stepping into a compiled function, use the :Set Trap On Call command on the function in which you want to step, then use the :Single Step command.

### 3.7.7 Debugger Commands for System Transfer

The Debugger provides commands that allow you to enter other systems while debugging. These systems are:

Zmacs, which allows you to edit your function

A mail message window, which allows you to mail a bug report

The Window Debugger

The Debugger commands that transfer you to these other systems are:

:Edit Function (c-E)

:Mail Bug Report (c-M)

:Window Debugger (c-n-W)

### Edit Function Command

:Edit Function *function*

c-E

Enters the Zmacs editor to bring up the current function or any other function for editing. This command lets you look at the function's source code. This is useful when you have found the function that caused the error and want to fix the code right away. The editor command c-z returns to the Debugger, if it is still there.

*function*

A stack frame that you select with the mouse or a function spec that specifies which function you want to edit. (Default edits the current function.)

*Suggested mouse operations*

To edit a function: Point the mouse at the function's stack frame and press `m-Mouse-Left`.

**Mail Bug Report Command**

`:Mail Bug Report keyword` `c-M`

Brings up a mail message window and puts a backtrace into a mail message to be mailed as a bug report.

This command creates a new process and runs the `bug` function in that process. It starts up a mail-sending window that contains a copy of the error message and a detailed backtrace of the stack. You are expected to report information explaining what you were doing when the problem occurred, preferably including a way for the person reading the bug report to make the problem happen again. The stack trace by itself is not adequate information for debugging. When you type the `END` key, the bug report is sent as mail, and you are brought back into the Debugger.

While composing the bug report, you can use normal window-switching commands such as `FUNCTION S` to switch back and forth between the Debugger and the mail-sending window.

A numeric argument given with this command's accelerator, `c-M`, as well as the `:Nframes` keyword, specifies the number of stack frames to put in your bug report; for example, `c-5 c-M` puts five frames into your report. The current stack frame begins the backtrace, so you might want to enter the `:Top Of Stack` command before you use `:Mail Bug Report`, if you have been examining frames other than the one that got the error. `:Top Of Stack` makes sure the error frame begins the backtrace.

*keyword*                    `:Nframes`

`:Nframes`                    `{stack-frame, number}` Specifies the number of stack frames to put into your bug report. Select a *stack-frame* with the mouse, or enter the *number* of most recent stack frames you want to send in your bug report. Frames that you specify show detailed information in the mail message. (Default places eight most recent frames into the mail message.)

*Suggested mouse operations*

- To put a backtrace in a mail message: Display the backtrace with the `:Show Backtrace` command, point the mouse at the *last* frame you want included in

your backtrace, and click `Mouse-Left`. All frames up to and including the frame you clicked on are put into the mail message.

### Window Debugger Command

`:Window Debugger` `c-m-w`

Enters the Window Debugger.

### 3.7.8 Miscellaneous Debugger Commands

There are a few miscellaneous Debugger commands that do not fit into any logical category. These commands are:

`:Help` (`c-HELP`)

`:Set Stack Size`

#### Help Command

`:Help` `c-HELP`

Displays a list of all available Debugger commands with brief descriptions and key-binding accelerators.

#### Set Stack Size Command

`:Set Stack Size` *stack-type stack-size*

Sets the size of a stack.

*stack-type*           The type of the stack. Enter Control, Binding, or Data.  
(Default is Control.)

*stack-size*           The size of the stack. Enter a number of machine words that  
represents the stack size.

## 3.8 Debugger Functions

The Debugger's evaluation environment lets you type in Lisp forms, which it reads and evaluates in the lexical context of the current frame, and then prints. When you are typing these forms, you can use the following functions to examine or modify the arguments, local variables, function object, and values being returned in the current frame.

- dbg:arg** *name-or-number* *Function*  
 Returns the value of argument *name-or-number* in the current stack frame. **(setf (dbg:arg *n*) *x*)** sets the value of the argument *n* in the current frame to the value of *x*. *name-or-number* can be the number of the argument (for example, 0 to specify the first argument) or the name of the argument. This function can be called only from the Debugger's evaluation environment.
- dbg:loc** *name-or-number* *Function*  
 Returns the value of the local variable *name-or-number* in the current stack frame. **(setf (dbg:loc *n*) *x*)** sets the value of the local variable *n* in the current frame to the value of *x*. *name-or-number* can be the number of the local variable (for example, 0 to specify the first local variable) or the name of the local variable. This function can be called only from the Debugger's evaluation environment.
- dbg:fun** *Function*  
 Returns the function object of the current stack frame. **(setf (dbg:fun) *x*)** sets the function object of the current frame to the value of *x*. This function can be called only from the Debugger's evaluation environment.
- dbg:val** &optional *val-no* 0 *Function*  
 Returns the value of the *val-no*th value to be returned from the current stack frame. **(setf (dbg:val *val-no*) *x*)** sets the value of the *val-no*th value to be returned from the current frame to the value of *x*. *val-no* must be a fixnum (since values do not have names) and defaults to 0. **(dbg:val)** without a value number gives the first value. This function can be called only from the Debugger's evaluation environment.
- dbg:monitor-location** (*location* &key (*read* nil) (*write* t) (*makunbound* (eq *write* t)) (*boundp* (eq *read* t)) *locate* *name*) *Function*  
 Monitors a location; that is, causes entry to the Debugger whenever *location* is accessed by a process. *location* is a locative to the location to be monitored; for example, (zl:value-cell-location 'foo). Descriptions of other arguments follow:

*read* {**t**, **nil**} monitors the location for reads. (Default is **nil**.)

*write* {**t**, **nil**} monitors the location for writes. (Default is **t**.)

*makunbound* {**t**, **nil**} monitors the location for **makunbound** operations. (Default is the value of *write*)

*boundp* {**t**, **nil**} monitors the location for **boundp** operations. (Default is the value of *read*.)

*locate* {**t**, **nil**} monitors the location for **locf** operations. (Default is **nil**.)

**dbg:monitor-instance-variable** *instance instance-variable-name &key* *Function*  
(*read nil*) (*write t*) *makunbound boundp locate*

Monitors an instance variable; that is, causes entry to the Debugger whenever the instance variable is accessed by a process. *instance* is the name of an instance containing an *instance-variable* you want to monitor. Descriptions of other arguments follow:

*read* {**t**, **nil**} monitors the instance variable for reads. (Default is **nil**.)

*write* {**t**, **nil**} monitors the instance variable for writes. (Default is **t**.)

*makunbound* {**t**, **nil**} monitors the instance variable for **makunbound** operations. (Default is **nil**.)

*boundp* {**t**, **nil**} monitors the instance variable for **boundp** operations. (Default is **nil**.)

*locate* {**t**, **nil**} monitors the instance variable for **locf** operations. (Default is **nil**.)

**dbg:unmonitor-location** *location* *Function*  
Unmonitors a location. *location* is a locative to the location you want to stop monitoring.



### 3.9 Debugger Variables

The Debugger uses the following variables:

**dbg:\*frame\*** *Variable*

Inside the Debugger's evaluation environment, the value of **dbg:\*frame\*** is the location of the current frame.

**dbg:\*defer-package-dwim\*** *Variable*

When this is **nil** (the default), the Debugger searches over all packages to find any look-alike symbols when errors concerning unbound variables occur.

When the option is not **nil**, the search does not occur until you press **c-sh-P**. In this case, the Debugger offers **c-sh-P** in the list of commands even if the search would find no look-alike symbols.

**dbg:\*debug-io-override\*** *Variable*

This is used during debugging to divert the Debugger to a stream that is known to work. If the value of this variable is **nil** (the default), the Debugger uses the stream that is the value of **zl:debug-io**. But if the value of **dbg:\*debug-io-override\*** is not **nil**, the Debugger uses the stream that is the value of this variable instead. This variable should always be set (using **setq**), not bound, so all processes and stack groups can see it.

**dbg:\*show-backtrace\*** *Variable*

Backtrace information appears when you enter the Debugger. The default is **nil**.

<i>Value</i>	<i>Meaning</i>
<b>nil</b>	The Debugger startup message does not include any backtrace information.
<b>t</b>	The Debugger startup message includes a three-element backtrace.

## 4. Summary of Debugger Commands

The following table summarizes all Debugger commands in alphabetical order. For each command, the table lists the command name, accelerators, positional arguments, keywords, and useful mouse operations.

This table appears only in the printed book. It does not appear online, in the Document Examiner. In the Document Examiner, you will see just an alphabetical list of all commands and their accelerators. To view a full command description for any command, simply point the mouse at the desired command in this list, and click `Mouse-Left`.

See the section "Debugger Command Descriptions", page 29. See also the online help file by pressing `c-HELP`.

COMMAND	ARGUMENTS	KEYWORDS	MOUSE
:Abort ABORT, c-Z	—	—	—
:Analyze Frame c-m-Z	—	—	—
:Bottom Of Stack m->	—	:Detailed {Yes, No}	—
:Clear All Breakpoints	<i>compiled-function-spec</i>	—	—
:Clear Breakpoint	<i>compiled-function</i> <i>pc</i>	—	On a code fragment or PC instruction: c-m-Mouse-Middle
:Clear Trap On Call c-X c-C	—	—	—
:Clear Trap On Exit c-X c-E	—	:All {Yes, No}	—
:Describe Last c-m-D	—	—	On any object: Mouse-Middle

COMMAND	ARGUMENTS	KEYWORDS	MOUSE
:Disable Aborts c-X T	—	—	—
:Disable Condition Tracing c-X T	—	—	—
:Edit Function c-E	<i>function</i>	—	On a function spec: m-Mouse-Left
:Enable Aborts	—	—	—
:Enable Condition Tracing c-X T	<i>condition</i> {t, nil, <i>condition</i> }	:Conditional {Always, Mode-Lock, Never, Once}	—
:Find Frame c-S	<i>string</i>	:Detailed {Yes, No} :Reverse {Yes, No}	—
:Help c-HELP	—	—	—
:Mail Bug Report c-M	—	:Nframes { <i>stack-frame</i> , <i>number</i> }	—
:Monitor Variable	<i>variable</i>	:Boundp {Yes, No} :Cell {Value-Cell, Function-Cell} :Locf {Yes, No} :Makunbound {Yes, No} :Read {Yes, No} :Write {Yes, No}	On a structure slot or instance variable: c-m-sh-Mouse-Left
:Next Frame LINE, c-N, m-N, c-m-N	—	:Detailed {Yes, No} :Internal {Yes, No} :Nframes { <i>number</i> }	—

COMMAND	ARGUMENTS	KEYWORDS	MOUSE
:Previous Frame RETURN, c-P, m-P, c-m-P, c-m-U	—	:Detailed {Yes, No} :Internal {Yes, No} :Nframes { <i>number</i> } :To Interesting {Yes, No}	—
:Proceed RESUME	—	—	—
:Proceed Trap On Call c-X m-C	—	—	—
:Reinvoke c-m-R	—	:New Args {Yes, No}	—
:Restart Trap On Call c-X c-m-C	—	—	—
:Return c-R	—	—	—
:Set Breakpoint	<i>compiled-function</i> <i>pc</i>	:Action {Show-All, Show-Args, Show-Locals, <i>expression</i> }  :Conditional {Always, Mode-Lock, Never, Once, <i>expression</i> }	On a code fragment or PC: c-m-Mouse-Left
:Set Current Frame	<i>stack-frame</i>	—	On a stack frame: Mouse-Left
:Set Stack Size	<i>stack-type</i> <i>stack-size</i>	—	—

COMMAND	ARGUMENTS	KEYWORDS	MOUSE
:Set Trap On Call c-X C	—	—	—
:Set Trap On Exit c-X E	—	:All {Yes, No}	—
:Show Arglist c-X c-A	—	—	—
:Show Argument c-m-A	<i>argument</i> { <i>number</i> , All}	—	—
:Show Backtrace c-B, m-B, c-m-B	—	:Detailed {Yes, No} :Internal {Yes, No} :Nframes { <i>number</i> }	—
:Show Bindings c-X B	—	:All {Yes, No} :Matching { <i>string</i> }	—
:Show Breakpoints	—	—	—
:Show Catch Blocks	—	:All {Yes, No}	—
:Show Compiled Code c-X D	<i>compiled-function-spec</i> <i>from-pc</i> <i>to-pc</i>	—	Set a breakpoint on a PC: c-m-Mouse-Left  Clear a breakpoint on a PC: c-m-Mouse-Middle
:Show Condition Handlers	—	:All {Yes, No}	—
:Show Frame REFRESH, c-L, m-L	—	:Clear Window {Yes, No} :Detailed {Yes, No}	—

COMMAND	ARGUMENTS	KEYWORDS	MOUSE
:Show Function c-m-F	—	—	—
:Show Instruction c-m-l	—	—	—
:Show Lexical Environment	—	—	—
:Show Local c-m-L	<i>local-variable</i> { <i>number</i> , All}	—	—
:Show Monitored Locations	—	—	—
:Show Proceed Options	—	—	On an option: Mouse-Left
:Show Rest Argument	—	—	—
:Show Source Code c-X c-D	<i>compiled-function-spec</i>	—	Set a breakpoint on a code fragment: c-m-Mouse-Left  Clear a breakpoint on a code fragment: c-m-Mouse-Middle  Evaluate a form: m-Mouse-Middle
:Show Special	<i>symbol</i>	:Environment {Program, Debugger, Streams}	—
:Show Stack	—	—	—
:Show Standard Value Warnings	—	—	—

COMMAND	ARGUMENTS	KEYWORDS	MOUSE
:Show Value c-m-V	<i>value {number}</i>	—	—
:Single Step c-sh-S	—	—	—
:Symeval In Last Instance c-X c-l	<i>symbol</i>	—	—
:Throw c-T	<i>symbol</i> <i>form</i>	—	—
:Top Of Stack m-<	—	:Detailed {Yes, No}	—
:Unmonitor Variable	<i>symbol</i> { <i>location</i> , RETURN}	:Cell {Value-Cell Function-Cell}	On a structure slot or instance variable: c-m-sh- Mouse-Middle
:Use Dynamic Environment c-X I	—	—	—
:Use Lexical Environment c-X I	—	—	—
:Window Debugger c-m-W	—	—	—

## 5. Tracing Function Execution

The trace facility allows you to *trace* some functions. Tracing is useful when you need to find out why a program behaves in an unexpected manner, particularly when you suspect that arguments are being passed incorrectly or functions are being called in the wrong sequence. The Trace facility is closely compatible with Maclisp.

Certain special actions are taken when a traced function is called and when it returns. The default tracing action prints a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and values.

You invoke the trace facility in several ways:

- Use the **trace** and **untrace** special forms.
- Click on [Trace] in the System menu. Enter or point to the function to be traced; a menu of options pops up.
- Invoke the Trace (M-X) command in the editor. Enter the function to be traced; a menu of options pops up.

The menu options are also available with **trace**; however, the syntax is complex.

**trace**

*Special Form*

A trace form looks like:

```
(trace spec-1 spec-2 ...)
```

Each *spec* can take any of the following forms:

a symbol

This is a function name, with no options. The function is traced in the default way, printing a message each time it is called and each time it returns.

a list (*function-name option-1 option-2 ...*)

*function-name* is a symbol and the *options* control how it is to be traced. For a list of the various options: See the section "Options To trace", page 74. Some options take arguments, which should be given immediately following the option name.

a list (**:function** *function-spec option-1 option-2 ...*)

This option is like the previous form except that *function-spec* need not be a symbol. (See the section "Function Specs" in *Symbolics Common Lisp: Language Concepts*.) It exists because if



*function-name* were a list in the previous form, it would instead be interpreted as the following form:

a list ((*function-1 function-2...*) *option-1 option-2 ...*)

All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

**trace** returns as its value a list of names of all functions it traced. If called with no arguments, as just **(trace)**, it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, **trace** calls **untrace** before setting up the new trace.

Tracing is implemented with encapsulation, so if the function is redefined (for example, with **defun** or by loading it from a compiled code file) the tracing is transferred from the old definition to the new definition.

It is recommended that you trace only user-defined functions and avoid tracing the system functions. Although some of the background processes use these functions, they never expect to have to type out anything. If they do have to type out something, the process will hang until you let it type out.

See the section "Encapsulations" in *Symbolics Common Lisp: Language Concepts*.

See the section "Options To trace", page 74.

## 5.1 Options To trace

The following **trace** options exist:

### **:break** *pred*

Enters a Dynamic Lisp Listener break loop after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-nil. During the break, the symbol **arglist** is bound to a list of the arguments of the function.

### **:exitbreak** *pred*

This is just like **:break** except that the break loop is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the break, the symbol **arglist** is bound to a list of the arguments of the function, and the symbol **values** is bound to a list of the values that the function is returning.

**:error** Calls the Debugger when the function is entered. Use **RESUME** to continue

execution of the function. If this option is specified, no printed trace output appears other than the error message displayed by the Debugger. (Note: If you also want to call the Debugger when the function returns, use the Debugger's `:Set Trap On Exit (c-X E)` command.)

**:step** Steps through interpreted code of a function whenever the function is called. For compiled code, use the Debugger's `:Single Step` command. See the section "Single Step Command", page 61.

See the section "Stepping Through an Evaluation", page 85.

**:entrycond *pred***

Prints trace information on function entry only if *pred* evaluates to non-**nil**.

**:exitcond *pred***

Prints trace information on function exit only if *pred* evaluates to non-**nil**.

**:cond *pred***

Prints trace information on function entry and exit only if *pred* evaluates to non-**nil**.

**:wherein *function***

Traces the function only when it is called, directly or indirectly, from the specified function *function*. You can give several trace specs to `trace`, all specifying the same function but with different `:wherein` options, so that the function is traced in different ways when called from different functions.

This is different from `advise-within`, which only affects the function being advised when it is called directly from the other function. The `trace :wherein` option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, and so on.

**:per-process *process***

Traces the function in the specified process only. It pops up a menu of processes and you choose the one in which to trace the function.

**:argpdl *pdl***

Specifies a symbol *pdl*, whose value is initially set to **nil** by `trace`. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments are pushed onto the *pdl* when the function is entered, and then popped when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own *pdl*, or one *pdl* can serve several functions.

**:entryprint** *form*

*form* is evaluated and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values will appear, preceded by \.

**:exitprint** *form*

*form* is evaluated and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values will appear, preceded by \.

**:print** *form*

*form* is evaluated and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values will appear, preceded by \.

**:entry** *list*

Specifies a list of arbitrary forms whose values are printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \ to separate it from the other information.

**:exit** *list*

Similar to **:entry**, but specifies expressions whose values are printed with the exit-trace. The list of values printed is preceded by \.

**:arg** **:value** **:both** **nil**

Specifies which of the usual trace printouts should be enabled.

<i>If you specify</i>	<i>Then</i>
<b>:arg</b>	On function entry prints the name of the function and the values of its arguments.
<b>:value</b>	On function exit prints the returned value(s) of the function.
<b>:both</b>	Same as if both <b>:value</b> and <b>:arg</b> were specified.
<b>nil</b>	Same as if neither <b>:value</b> or <b>:arg</b> was specified.
None	The default is to <b>:both</b> .

If any further *options* appear after one of these, they are not treated as options. Rather, they are considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function, along with the normal trace information. The values printed are preceded by a //, and follow any values specified by **:entry** or **:exit**. Note that since these options "swallow" all following options, if one is given it should be the last option specified.

If the variable **arglist** is used in any of the expressions given for the **:cond**, **:break**, **:entry**, or **:exit** options, or after the **:arg**, **:value**, **:both**, or **nil** option, when those expressions are evaluated the value of **arglist** will be bound to a list of the arguments given to the traced function. Thus the following form would cause a break in **foo** if and only if the first argument to **foo** is **nil**.

```
(trace (foo :break (null (car arglist))))
```

If the **:break** or **:error** option is used, the variable **arglist** will be valid inside the break-loop. If you **setq arglist**, the arguments seen by the function will change.

Similarly, the variable **values** will be a list of the resulting values of the traced function. For obvious reasons, this should only be used with the **:exit** option. If the **:exitbreak** option is used, the variables **values** and **arglist** are valid inside the break-loop. If you **setq values**, the values returned by the function will change.

You can "factor" the trace specifications, as explained earlier. For example,

```
(trace ((foo bar) :break (bad-p arglist) :value))
```

is equivalent to

```
(trace (foo :break (bad-p arglist) :value)
       (bar :break (bad-p arglist) :value))
```

Since a list as a function name is interpreted as a list of functions, nonatomic function names are specified as follows:

```
(trace (:function (:method flavor :message) :break t))
```

(See the section "Function Specs" in *Symbolics Common Lisp: Language Concepts*.)

### **sys:trace-compile-flag**

*Variable*

If the value of **trace-compile-flag** is non-**nil**, the functions created by **trace** will get compiled, allowing you to trace special forms such as **cond** without interfering with the execution of the tracing functions. The default value of this flag is **nil**.

## **5.2 Controlling the Format Of trace Output**

Tracing output is printed on the stream that is the value of **trace-output**. This is synonymous with **terminal-io** unless you change it. Following is an example of the default form of **trace** output:

```

1 Enter FACT 4.
| 2 Enter FACT 3.
| 3 Enter FACT 2.
| | 4 Enter FACT 1.
| | 5 Enter FACT 0.
| | 5 Exit FACT 1.
| | 4 Exit FACT 1.
| 3 Exit FACT 2.
| 2 Exit FACT 6.
1 Exit FACT 24.

```

You can use the variables `si:*trace-columns-per-level*`, `si:*trace-bar-p*`, `si:*trace-bar-rate*`, and `si:*trace-old-style*` to control the format of `trace` output.

**si:\*trace-columns-per-level\*** *Variable*

For `trace` output, controls the number of columns of indentation that are added for each level of function call. The value must be an integer. The default is 2.

**si:\*trace-bar-p\*** *Variable*

For `trace` output, controls whether columns of vertical bars are printed. If the value is not `nil`, they are printed; otherwise, spaces are printed instead of the vertical bars. The default is `t` (print the bars).

**si:\*trace-bar-rate\*** *Variable*

When `si:*trace-bar-p*` is not `nil`, columns of vertical bars are printed in `trace` output for every  $n$  levels of function call, where  $n$  is the value. The value must be an integer. The default is 2.

**si:\*trace-old-style\*** *Variable*

If not `nil`, the old, Maclisp-compatible form of printing `trace` output is used. The default is `nil` (use the new style).

### 5.3 Untracing Function Execution

**untrace** `&quote &rest fns` *Special Form*

Use `untrace` to undo the effects of `trace` and restore functions `fns` to their normal, untraced state. `untrace` takes multiple specifications, for example, `(untrace foo bar baz)`. Calling `untrace` with no arguments untraces all functions currently being traced.

## 6. Advising a Function

To *advise* a function is to tell a function to do something extra in addition to its actual definition. Advising is achieved by means of the function **advise**. The something extra is called a piece of advice, and it can be done before, after, or around the definition itself. The advice and the definition are independent, in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function to give the user information about when and how the function is called and when and with what value it returns. Advising is intended for semipermanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing out a change to a function in a way that is easy to retract. In this case, you would call **advise** from the console. It can also be used for customizing a function that is part of a program written by someone else. In this case you would be likely to put a call to **advise** in one of your source files or your login init file rather than modifying the other person's source code. See the section "Logging In" in *User's Guide to Symbolics Computers*.

Advising is implemented with encapsulation, so if the function is redefined (for example, with **defun** or by loading it from a compiled code file), the advice will be transferred from the old definition to the new definition. See the section "Encapsulations" in *Symbolics Common Lisp: Language Concepts*.

**advise** *function class name position &body forms* *Special Form*

A function is advised by the special form

(*advise function class name position  
form1 form2...*)

None of this is evaluated.

*function* Specifies the function to put the advice on. It is usually a symbol, but any function spec is allowed. (See the section "Function Specs" in *Symbolics Common Lisp: Language Concepts*.)

*class* Specifies either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). For more information about the meaning of **:around**, **:before**, and **:after** advice: See the section "**:around** Advice", page 82.

*name* Specifies an arbitrary symbol that is remembered as the name of this particular piece of advice. It is used to keep track of multiple pieces of advice on the same function. If you have no name in mind, use `nil`; then we say the piece of advice is anonymous.

A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first.

Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

*position* Specifies where to put this piece of advice in relation to others of the same class already present on the same function.

Position can have these values:

- *position* can be `nil`. The new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.
- *position* can be a number, which is the number of pieces of advice of the same class to precede this one. For example, 0 means at the beginning; a very large number means at the end.
- *position* can have the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

*forms* Specifies the advice; they get evaluated when the function is called.

**Example:** The following form modifies the factorial function so that if it is called with a negative argument it signals an error instead of running forever.

```
(advise factorial :before negative-arg-check nil
  (if (minusp (first arglist))
      (ferror "factorial of negative argument"))))
```

**unadvise** &optional *function class position* *Special Form*

Removes pieces of advice. None of its subforms are evaluated. *function* and *class* have the same meaning as they do in the function **advise**. *position* specifies which piece of advice to remove. It can be the numeric index (0 means the first one) or it can be the name of the piece of advice.

**unadvise** can remove more than one piece of advice if some of its arguments are missing or **nil**. The arguments *function*, *class*, and *position* all act independently. A missing value or **nil** means all possibilities for that aspect of advice. For example, the following form removes all **:before**, **:after**, and **:around** advice named **negative-arg-check** on the **factorial** function:

```
(unadvise factorial nil negative-arg-check)
```

In this example **unadvise** removes all **:around** advice on all functions in all positions with all names:

```
(unadvise nil :around)
```

In this example **unadvise** removes all classes of advice named **my-personal-advice** on all functions:

```
(unadvise nil nil my-personal-advice)
```

(**unadvise**) removes all advice on all functions, since *function*, *class*, and *position* take on all possible values.

The following are the primitive functions for adding and removing advice. Unlike the special forms **advise** and **unadvise**, the following are functions and can be conveniently used by programs. **advise** and **unadvise** are actually macros that expand into calls to these two.

**si:advise-1** *function class name position forms* *Function*

Adds advice. The arguments have the same meaning as in **advise**. Note that the *forms* argument is *not* a **&rest** argument.

**si:unadvise-1** *function &optional class position* *Function*

Removes advice. *function*, *class*, and *position* are independent. If *function*, *class*, or *position* is **nil**, or if *class* or *position* is unspecified, all classes of advice or advice for all functions, at all positions, or with all names is removed.

You can find out manually what advice a function has with **grindef**, which grinds the advice on the function as forms that are calls to **advise**. These are in addition to the definition of the function.

To poke around in the advice structure with a program, you must work with the encapsulation mechanism's primitives. See the section "Encapsulations" in *Symbolics Common Lisp: Language Concepts*.



**si:advised-functions***Variable*

A list of all functions that have been advised.

## 6.1 Designing the Advice

For advice to interact usefully with the definition and intended purpose of the function, it must be able to interface to the data flow and control flow through the function. The system provides conventions for doing this.

The list of the arguments to the function can be found in the variable **arglist**. **:before** advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with:

```
(setq arglist (copylist arglist))
```

After the function's definition has been executed, the list of the values it returned can be found in the variable **values**. **:after** advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a **prog**, so any piece of advice can exit the entire function and return some values with **return**. No further advice will be executed. If a piece of **:before** advice does this, then the function's definition will not even be called.

## 6.2 :around Advice

A piece of **:before** or **:after** advice is executed entirely before or entirely after the definition of the function. **:around** advice is wrapped around the definition; that is, the call to the original definition of the function is done at a specified place inside the piece of **:around** advice. You specify where by putting the symbol **:do-it** in that place.

For example, **(+ 5 :do-it)** as a piece of **:around** advice would add 5 to the value returned by the function. This could also be done by the following:

```
(setq values (list (+ 5 (car values))))
```

as **:after** advice.

When there is more than one piece of **:around** advice, they are stored in a sequence just like **:before** and **:after** advice. Then, the first piece of advice in the sequence is the one started first. The second piece is substituted for **:do-it** in the first one. The third one is substituted for **:do-it** in the second one. The original definition is substituted for **:do-it** in the last piece of advice.



To remove advice from (:within bar foo), you can use **unadvise** on that function specifier. Alternatively, you can use **unadvise-within**.

**unadvise-within** *within-function* &optional *advised-function class position*      *Special Form*

An **unadvise-within** form looks like this:

(unadvise-within *within-function function-to-advise class position*)

It removes advice that has been placed on (:within *within-function function-to-advise*). The arguments *class* and *position* are interpreted as for **unadvise**.

For example, if those two arguments are omitted, then all advice placed on *function-to-advise* within *within-function* is removed. Additionally, if *function-to-advise* is omitted, all advice on any function within *within-function* is removed. If there are no arguments, then all advice on one function within another is removed. Other pieces of advice, which have been placed on one function and not limited to within another, are not removed.

(**unadvise**) removes absolutely all advice, including advice for one function within another.

The function versions of **advise-within** and **unadvise-within** are called **si:advise-within-1** and **si:unadvise-within-1** respectively. **advise-within** and **unadvise-within** are macros that expand into calls to the other two.

## 7. Stepping Through an Evaluation

The step facility gives you the ability to follow every step of the evaluation of an interpreted form and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. Use the step facility if your program is behaving strangely, and it is not obvious how it is getting into this strange state.

You can enter the stepper in two ways:

- Use the `zl:step` function.
- Use the `:step` option of `trace`.

### `zl:step` form

*Function*

`zl:step` evaluates *form* with single stepping. It returns the value of *form*.

For example, if you have a function named `foo`, and typical arguments to it might be `t` and `3`, you could say

```
(step '(foo t 3))
```

If a function is traced with the `:step` option, then whenever that function is called it will be single stepped. See the section "Options To `trace`", page 74. Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be handled by `zl:step`.

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a right-facing arrow (`→`) character. When a macro is expanded, the expansion is printed out preceded by a double arrow (`↔`) character. When a form returns a value, the form and the values are printed out preceded by a left-facing arrow (`←`) character; if more than one value is being returned, an and-sign (`^`) character is printed between the values.

Since the forms can be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from you. A variety of commands exist to tell the stepper how to proceed, or to look at what is happening.

- |                         |   |
|-------------------------|---|
| <code>c-N</code> (Next) | Steps to the next thing. The stepper continues until the next thing to print out, and it accepts another command. |
| <code>SPACE</code>      | Goes to the next thing at this level. In other words, it continues  |

to evaluate at this level, but does not step anything at lower levels. In this way you can skip over parts of the evaluation that do not interest you.

- c-U (Up) Continues evaluating until we go up one level. Similar to the SPACE command; it skips over anything on the current level as well as lower levels.
- c-X (Exit) Exits; finishes evaluating without any more stepping.
- c-T (Type) Retypes the current form in full (without truncation).
- c-G (Grind) Grinds (that is, pretty-prints) the current form.
- c-E (Editor) Enters the editor.
- c-B (Breakpoint)  
This command puts you into a breakpoint (that is, a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:
- step-form** The current form.
- step-values** The list of returned values.
- step-value** The first returned value.
- You can change the values of these variables within the current environment.
- You can also refer to local variables and arguments in the function.
- c-L Clears the screen and redisplay the last ten pending forms (forms being evaluated).
- m-L Like c-L, but does not clear the screen.
- c-m-L Like c-L, but redisplay all pending forms.
- ? or HELP Prints documentation on these commands.

It is strongly suggested that you write a little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

## 8. evalhook

The **evalhook** facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses **evalhook**; however, if you want to write your own stepper or something similar, then use this primitive albeit complex facility to do so.

### **evalhook**

*Variable*

If the value of **evalhook** is non-**nil**, then special things happen in the evaluator. When a form (any form, even a number or a symbol) is to be evaluated, **evalhook** is bound to **nil** and the function that was **evalhook**'s value is applied to one argument – the form that was trying to be evaluated. The value it returns is then returned from the evaluator.

**evalhook** is bound to **nil** by **zl:break** and by the Debugger, and **setqed** to **nil** when errors are dismissed by throwing to the Lisp top-level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on **evalhook**. It applies only to evaluation – whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function **eval**. It does *not* have any effect on compiled function references, on use of the function **zl:apply**, or on the "mapping" functions. (In Zetalisp, as opposed to Maclisp, it is not necessary to do (**zl:\*rset t**) nor (**zl:ssstatus evalhook t**). Also, Maclisp's special-case check for **zl:store** is not implemented.)

### **evalhook form evalhook &optional applyhook env**

*Function*

**evalhook** is a function that helps exploit the **evalhook** feature. The *form* is evaluated with **evalhook** lambda-bound to the function *evalhook*. The checking of **evalhook** is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger. *env* is used as the lexical environment for the operation. *env* defaults to the null environment.

Example:

```
;; This function evaluates a form while printing debugging
;; information.
(defun hook (x)
  (terpri)
  (evalhook x 'hook-function))
```

```
;; Notice how this function calls evalhook to evaluate the
;; form f, so as to hook the subforms.
(defun hook-function (f)
  (let ((v (evalhook f 'hook-function)))
    (format t "form: ~s~%value: ~s~%" f v)
    v))

;; This isn't a very good program, since if f returns multiple
;; values, it will not work.
```

The following output might be seen from `(hook '(cons (car '(a . b)) 'c))`:

```
form: (quote (a . b))
value: (a . b)
form: (car (quote (a . b)))
value: a
form: (quote c)
value: c
(a . c)
```

Normally after `eval` has evaluated the arguments to a function, it calls the function. If `applyhook` exists, however, `eval` calls the hook with two arguments: the function and its list of arguments. The values returned by the hook constitute the values for the form. The hook could use `zl:apply` on its arguments to do what `eval` would have done normally. This hook is active for special forms as well as for real functions.

Whenever either an `evalhook` or `applyhook` is called, both hooks are bound off. The `evalhook` itself can be `nil` if only an `applyhook` is needed.

`applyhook` catches only `zl:apply` operations done by `eval`. It does not catch `zl:apply` called in other parts of the interpreter or `zl:apply` or `funcall` operations done by other functions such as `mapcar`. In general, such uses of `zl:apply` can be dealt with by intercepting the call to `mapcar`, using the `applyhook`, and substituting a different first argument.

The argument list is like an `&rest` argument: it might be stack-allocated but is not guaranteed to be. Hence you cannot perform side-effects on it and you cannot store it in any place that does not have the same dynamic extent as the call to `applyhook`.

## 8.1 applyhook

`applyhook` provides a hook into `zl:apply`, much as `evalhook` provides a hook into `eval`.

**applyhook***Variable*

When the value of this variable is not **nil** and **eval** calls **zl:apply**, **applyhook** is bound to **nil** and the function that was its value is applied to two arguments: the function that **eval** gave to **zl:apply** and the list of arguments to that function. The value it returns is returned from the evaluator.

**applyhook** *function args evalhook applyhook &optional env**Function*

*function* is applied to *args* with **evalhook** lambda-bound to the function *evalhook* and with **applyhook** lambda-bound to the function *applyhook*. Like the **evalhook** function, this bypasses the first place where the relevant hook would normally be triggered. *env* is used as the lexical environment for the operation. *env* defaults to the null environment. *evalhook* or *applyhook* can be **nil**.





## **PART II.**

### **Miscellaneous Debugging Aids**



## 9. The Inspector

### 9.1 How the Inspector Works

The Inspector is a window-oriented program for inspecting data structures. When you ask to inspect a particular object, its components are displayed. The particular components depend on the type of object; for example, the components of a list are its elements, and those of a symbol are its value binding, function definition, and property list.

The component objects displayed on the screen by the Inspector are mouse-sensitive, allowing you to do something to that object, such as inspect it, modify it, or give it as the argument to a function. Choose these operations from the menu pane at the top-right part of the screen.

When you click on a component object itself, that component object gets inspected. It expands to fill the window and its components are shown. In this way, you can explore a complex data structure, looking into the relationships between objects and the values of their components.

The Inspector can be part of another program or it can be used standalone; for example, the Window Debugger can utilize some of the panes of the Inspector. Note, however, that although the display looks the same as that of the standalone Inspector, the handling of the mouse buttons depends upon the particular program being run.

Figure 1 shows the standalone Inspector window. The display consists of the following panes, from top to bottom:

- A small interaction pane
- A history pane and menu pane
- Some number of inspection panes (three by default)

### 9.2 Entering and Leaving the Inspector

You can enter the standalone Inspector via:

- Select Activity *Inspector*
- `SELECT I`
- [Inspect] in the System menu

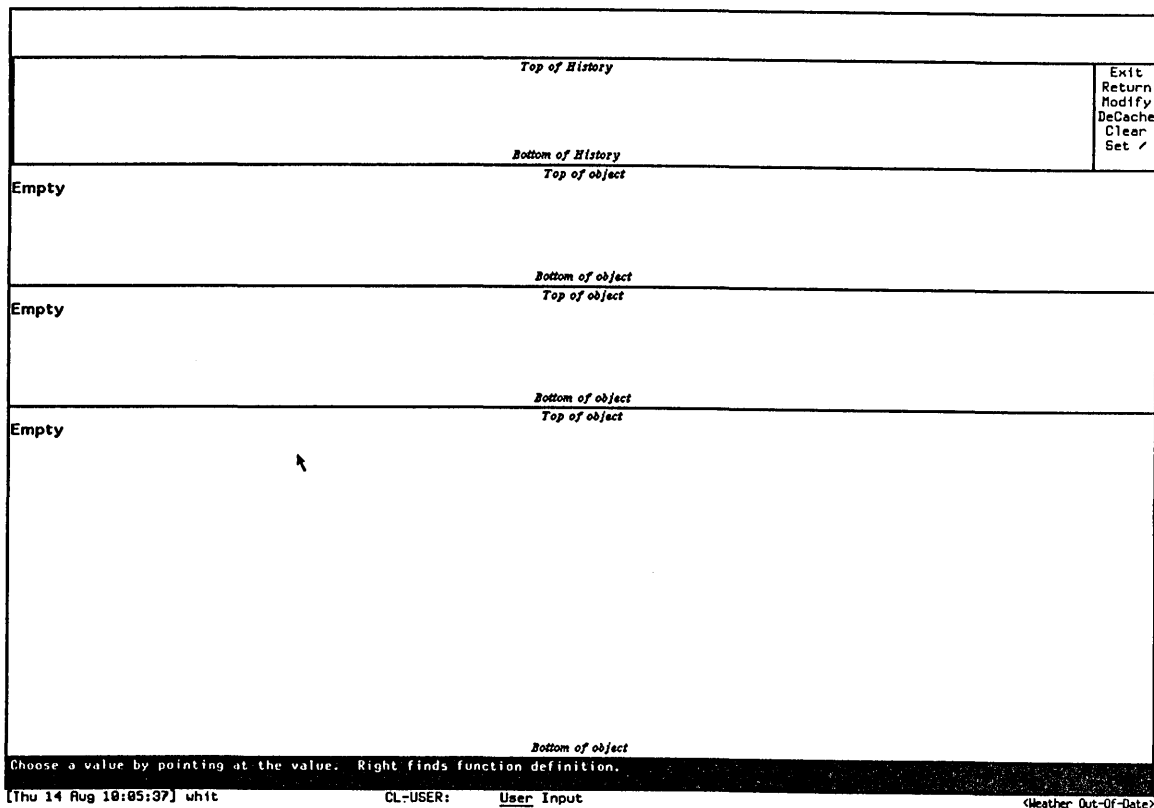


Figure 1. The Inspector

- The `Inspect` command, which inspects its argument, if any
- The `inspect` function, which inspects its argument, if any

Warning: If you enter with the `Inspect` command or the `inspect` function, the Inspector is not a separate activity from the Lisp Listener in which you invoke it. In this case you cannot use `SELECT L` to return to the Lisp Listener; you should *always* exit via the `[Exit]` or `[Return]` option in the Inspector menu. If you forget and exit the Inspector by selecting another activity, you might need to use `c-m-ABORT` to return the Lisp Listener to its normal state.

### 9.3 The Inspector Interaction Pane

The interaction pane has two functions: to prompt you and to receive input. If you are not being asked a question, then a read-eval-inspect loop is active. Any forms you type are echoed in the interaction pane and evaluated. The result is not printed, but rather inspected. When you are prompted for input, usually due to having invoked a menu operation, any input you type at the read-eval-inspect loop is saved away and erased from the interaction pane. When the interaction is finished, the input is re-echoed and you can continue to type the form.

### 9.4 The Inspector History Pane

The history pane maintains a list of all objects that you have inspected, allowing you to back up and continue down another path. The last recently displayed object is at the top of the list, and the most recently displayed object is at the bottom.

You can inspect any mouse-sensitive object in the history pane by clicking on it. In addition, you can perform other operations by placing the mouse cursor in the *line region*, which is the left-hand side of the history pane, the area bounded by the margin on one side and the list of objects on the other. In the line region the shape of the mouse cursor changes to a rightward-pointing arrow.

- Clicking left in the line region inspects the object. This is sometimes useful when the object is a list and it is inconvenient to position the mouse at the open parenthesis.
- Clicking middle deletes the object from the history.

The history pane also maintains a cache allowing quick redisplay of previously displayed objects. This means that merely reinspecting an object does not reflect any changes in its state. Clicking middle in the line region deletes the object from the cache as well as deleting it from the history pane. Use [DeCache] in the menu pane to clear everything from the cache.

The history pane has a scroll bar at the far left, as well as scrolling zones in the middle of its top and bottom edges. The last three lines of the history are always the objects being inspected in the inspection panes.

### 9.5 The Inspector Menu Pane

The menu pane (to the right of the history pane) displays these infrequently used but useful commands:

[Exit]	Equivalent to c-Z. Exits the Inspector and deactivates the frame.
[Return]	Similar to [Exit], but allows selection of an object to be returned as the value of the call to <b>inspect</b> .
[Modify]	Allows simple editing of objects. Selecting [Modify] changes the mouse sensitivity of items on the screen to only include fields that are modifiable. In the typical case of named slots, the names are the mouse-sensitive parts. When the field to modify has been selected, a new value can be specified either by typing a form to be evaluated or by using the mouse to select any normally mouse-sensitive object. The object being modified is redisplayed. Clicking right at any time aborts the modification.
[DeCache]	Flushes all knowledge about the insides of previously displayed objects and redisplayes the currently displayed objects.
[Clear]	Clears out the history, the cache, and all the inspection panes.
[Set] \	Sets the value of the symbol \ by choosing an object.

## 9.6 The Inspector Inspection Pane

Each inspection pane can inspect a different object. When you inspect an object it appears in the large inspection pane at the bottom, and the previously inspected objects shift upward.

At the top of an inspection pane is either a label, which is the printed representation of the object being inspected in that window, or the words "a list", which means a list is being inspected. The main body of an inspection pane is a display of the components of the object, labelled with their names, if any. You can scroll this display using the scroll bar on the left or the "more above" and "more below" scrolling zones at the top and bottom.

Clicking on any mouse-sensitive object in an inspection pane inspects that object. The three mouse buttons have distinct meanings, however.

- Clicking left inspects the object in the bottom pane, pushing the previous objects up.
- Clicking middle inspects the object but leaves the source (namely, the object being inspected in the window in which the mouse was clicked) in the second pane from the bottom.
- Clicking right tries to find and inspect the function associated with the selected object (for example, the function binding if a symbol was selected).

### 9.6.1 Inspection Pane Display

The information that the Inspector displays depends upon the type of the object:

Symbol	The name, value, function, property list, and package of the symbol are displayed. All but the name and the package are modifiable.
List	The list is displayed ground by the system grinder. Any piece of substructure is selectable, and any <code>car</code> or <code>atom</code> in the list can be modified.
Instance	The flavor of the instance, the method table, and the names and values of the instance-variable slots are displayed. The instance-variables are modifiable.
Hash Table	The flavor of the hash table, the method table, and the names and values of the instance-variable slots of the hash table are displayed, followed by the key/value pairs for the entries of the hash table. The value for a given key is modifiable.
Closure	The function, and the names and values of the closed variables are displayed. The values of the closed variables are modifiable.
Named structure	The names and values of the slots are displayed. The values are modifiable.
Array	The leader of the array is displayed if present. For one-dimensional arrays, the elements of the array are also displayed. The elements are modifiable.
Compiled code object	The disassembled code is displayed.
Select Method	The keyword/function pairs are shown, in alphabetical order by keyword. The function associated with a keyword is settable via the keyword.
Stack Frame	This is a special internal type used by the Display Debugger. It is displayed as either interpreted code (a list) or as a compiled code object with an arrow pointing to the next instruction to be executed.

## 9.7 Special Characters Recognized by the Inspector

Some special keyboard characters are recognized when not in the middle of typing in a form.



- c-Z Exits and deactivates the Inspector.
- BREAK Runs a break loop in the typeout window of the bottom-most inspection pane.
- ESCAPE Reads a form, evaluates it, and prints the result instead of inspecting it.

## 9.8 Examining a Compiled Code File

To examine a compiled code file, use **si:unbin-file**. The output format from **unbin-file** includes disassembled code for any compiled functions in the compiled code file.

**si:unbin-file** *file* &optional *outfile* *Function*  
Converts the compiled code file *file* to a human-readable file, which you can optionally specify. It includes disassembled code for any compiled functions in the compiled code file.

## 10. The Peek Program

### 10.1 Overview of Peek

You start up Peek by pressing SELECT P, by using the Select Activity Peek command, or by evaluating (**zl:peek**).

The Peek program gives a dynamic display of various kinds of system status. When you start up Peek, a menu is displayed at the top, with one item for each system-status mode. The item for the currently selected mode is highlighted in reverse video. If you click on one of the items with the mouse, Peek switches to that mode. Pressing one of the keyboard keys as listed in the Help message also switches Peek to the mode associated with that key. The Help message is a Peek mode; Peek starts out in this mode.

Pressing the HELP key displays the Help message.

The Q command exits Peek and returns you to the window from which Peek was invoked.

Most of the modes are dynamic: they update some part of the displayed status periodically. The time interval between updates can be set using the Z command. Pressing *n*Z, where *n* is some number, sets the time interval between updates to *n* seconds. Using the Z command does not otherwise affect the mode that is running.

Some of the items displayed in the modes are mouse sensitive. These items, and the operations that can be performed by clicking the mouse on them, vary from mode to mode. Often clicking the mouse on an item gives you a menu of things to do to that object.

The Peek window has scrolling capabilities, for use when the status display is longer than the available display area. SCROLL or c-V scrolls the window forward (towards the bottom), m-SCROLL or m-V scrolls it backward (towards the top).

As long as the Peek window is exposed, it continues to update its display. Thus a Peek window can be used to examine things being done in other windows in real time.

**zl:peek** &optional (*character* (quote tv:p)) *Function*  
zl:peek displays various information about the system, periodically updating it. It has several modes, which are entered by pressing a single key that is the name of the mode. The initial mode is selected by the argument, *character*. If no argument is given, **zl:peek** starts out by explaining what its modes are.

The Help message consists of the following:

This is the Peek utility program. It shows a continually updating display of status about some aspect of the system, depending on what mode it is in. The available modes are listed below. Each has a name, followed by a single character in parentheses, followed by a description. To put Peek into a given mode, click on the name of the mode, in the command menu above. Alternatively, type the single character shown below.

Processes (P):

Show all active processes, their states, priorities, quanta, idle times, etc.

Areas (A):

Show all the areas in virtual memory, their types, allocation, etc.

File System (F):

Show all of our connections to various file servers.

Windows (W):

Show all the active windows and their hierarchical relationships.

Servers (S):

Show all active network servers and what they are doing.

Network (N):

Show all local networks, their state and active connections, and network interfaces.

Help (<HELP>):

Explain how this program works.

Quit (Q):

Bury PEEK window, exiting PEEK

Hostat (H):

Show the status of all hosts on the Chaosnet

There are also the following single-character commands:

Z (preceded by a number): Set the amount of time between updates, in seconds.

By default, the display is updated every two seconds.

<SPACE>: Immediately update the display.

The commands P, A, F, W, S, H, and N each place you in a different Peek mode, to examine the status of different aspects of the Lisp Machine system.

## 10.2 Peek Modes

### Processes (P)

In Processes mode, invoked by pressing P or by clicking on the [Processes] menu item, you see all the processes running in your environment, one line for each. The process names are mouse sensitive; clicking on one of them pops up a menu of operations that can be performed:

#### Arrest (or Un-Arrest)

Arrest causes the process to stop immediately. Unarrest causes it to pick up where it left off and continue.

#### Flush

Causes the process to go into the state Wait Forever. This is one way to stop a runaway process that is monopolizing your machine and not responding to any other commands. A process that has been flushed can be looked at with the debugger or inspector and can be reset.

#### Reset

Causes the process to start over in its initialized state. This is one way to get out of stuck states when other commands do not work.

#### Kill

Causes the process to go away completely.

#### Debugger

Enters the Debugger to look at the process.

#### Describe

Displays information about the process.

#### Inspect

Enters the Inspector to look at the process.

See the section "Introduction to Processes" in *Internals, Processes, and Storage Management*.

### Areas (A)

Areas mode, invoked by pressing A or by clicking on [Areas], shows you information about your machine's memory. The first line is hardware information: the amount of physical memory on the machine, the amount of swapping space remaining in virtual memory, and how many wired pages of memory the machine has. The following lines show all the areas in virtual memory, one line for each. For each area you are shown how many regions it contains, what percentage of it is free, and the number of words (of the total) in use. Clicking on an area inserts detailed information about each region: its number, its starting address, its length, how many words are used, its type, and its GC status. See the section "Areas" in *Internals, Processes, and Storage Management*.

## Meters (M)

Meters mode, invoked by pressing M or by clicking on [Meters], shows you a list of all the metering variables for storage, the garbage collector, and the disk. There are two types of storage and disk meters:

- |        |   |
|--------|---|
| Timers | Timers have names that start with <b>zl-user:*ms-time-</b> and keep a total of the milliseconds spent in some activity.           |
| Counts | Counts have names that start with <b>zl-user:*count-</b> and keep a running total of the number of times some event has occurred. |

The garbage collector meters fall into two groups according to which part of the garbage collector they pertain to: the scavenger or the transporter. See the section "Operation of the Garbage Collector".

## File System (F)

File System mode, invoked by pressing F or by clicking on [File System], provides you information about your network connections for file operations. For each host the access path, protocol, user-id, host or server unit number, and connection state are listed. For active connections information about the actual packet flow is also given. The various items are mouse sensitive. For hosts, you can get hostat information, do a file reset, log in remotely, find out who is on the remote machine, and send a message to the machine. You can reset, describe, or inspect data channels, and close streams.

Resetting an access path makes the server on a foreign host go away, which might be useful to free resources on that host or if you suspect that the server is not working correctly.

## Windows (W)

Windows mode, invoked by pressing W or clicking on [Windows], shows you all the active windows in your environment with the panes they contain. This allows you to see the hierarchical structure of your environment. The items are mouse sensitive. Clicking on a window name pops up a menu of operations that you can perform on the window.

## Servers (S)

Clicking on [Servers] or pressing S puts Peek in Servers mode. If your machine is a server (for example, a file server), Servers mode shows the status of each active server.

## Network (N)

Network mode, invoked by pressing N or by clicking on [Network], shows information about the networks connected to your machine. For each network there are three headings for information:

### Active connections

The data channels that your machine has opened to another machine or machines on the network.

### Meters

Information about the data flow (packets) between your machine and other machines on the network.

### Routing table

A list of all the subnets and for each the route to take to send packets to a host on that subnet.

To view the information under one of these headings, you click on the heading. The hosts and data channels in the list of active connections are mouse sensitive. For hosts, you can get hostat information, do a file reset, login remotely, find out who is on the remote machine, and send a message to the machine. You can reset, describe, or inspect data channels.

Information about the hardware network interface is also displayed, as well as metering variables for the networks.

## Hostat (H)

Clicking on [Hostat] or pressing H starts polling all the machines connected to the local network. For each host on the network a line of information is displayed. Those machines that do not respond to the poll are marked as "Host not responding". You terminate the display by pressing c-ABORT.

## Help and Quit

Clicking on the [Help] menu item or pressing HELP displays the help information that is displayed when Peek is selected the first time.

Clicking on [Quit] or pressing Q buries the Peek window and returns you to the window from which you invoked Peek.



## **PART III.**

### **The Compiler**





## 11. Introduction to the Compiler

The purpose of the Symbolics Lisp compiler is to convert Lisp functions into programs in the Symbolics computer's instruction set. Compiled functions run more quickly and take up less storage than interpreted code. They are executed directly by the machine. The compiler checks for errors and issues warnings regarding faulty syntax, typographical errors, and undeclared variables. Because the compiler does all this checking, as well as the fact that compiling code does not lose any run-time checking, most users debug their programs in compiled form rather than debugging them in interpreted form and compiling them after they work.

### 11.1 How to Invoke the Compiler

You can invoke the compiler in several ways.

- Use one of several Zmacs commands to compile regions of Lisp code in an editor buffer to your Lisp environment. Some of the most common commands are Compile Region (M-X) (c-sh-C), Compile Changed Definitions of Buffer (M-X), and Compile Buffer (M-X). See the section "Compiling Lisp Programs in Zmacs" in *Text Editing and Processing*.
- Call the function `compile` to compile an interpreted function in the Lisp environment. Compiling an interpreted function in a Dynamic Lisp Listener converts the function into a compiled code object in memory. Programmers occasionally compile interpreted functions to examine the code generated by the compiler. To examine a compiled function in symbolic form, use the `disassemble` function.
- Use `compile-file` and related functions, Compile File (M-X), or Compile File at the Command Processor prompt to translate source files into compiled code files.
- Invoke `compile-system` or type Compile System at the Command Processor prompt to compile and load large programs, usually consisting of many files.



## 12. Structure of the Compiler

The Lisp compiler is actually composed of three distinct pieces of software:

- The stream compiler
- The function compiler
- The **bin** (binary) file dumper

The stream compiler accepts a stream of top-level Lisp forms and processes them. These forms are usually read from a stream of characters, which can be either a file or part or all of an editor buffer. The stream compiler passes forms recognized as function definitions through the function compiler. Certain other forms are also processed specially: See the section "How the Stream Compiler Handles Top-level Forms", page 111. Stream compiler output can be sent either to the Symbolics computer's virtual memory or to a file (via the **bin** file dumper) for later loading.

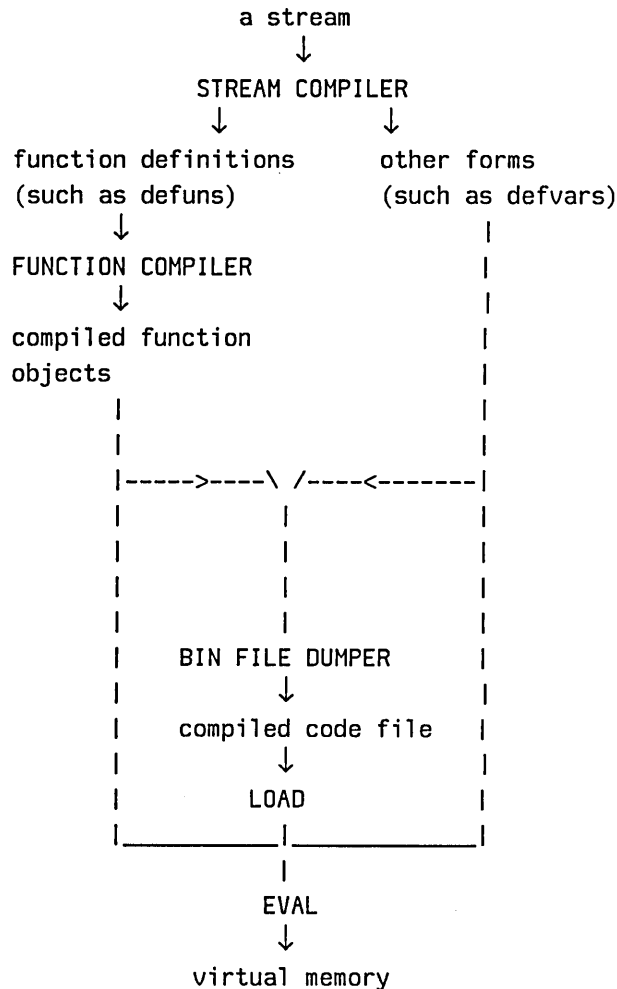
The function compiler takes a Lisp function and translates it from Lisp expressions into machine instructions. Its job includes expanding macros, performing optimizations, recognizing special forms, and recognizing calls to functions that have corresponding machine instructions. The function compiler is available to use by itself as the **compile** function; it is also called by the stream compiler.

The **bin** file dumper accepts a stream of Lisp forms and machine-instruction function definitions (compiled function objects) and writes them into a file in a compact form understood by the loading function (**zl:load**). The **bin** file dumper is available for use by itself as the **sys:dump-forms-to-file** function; it is also called by the stream compiler.

Different combinations of these compilers are available:

- The function compiler can be used by itself (via the **compile** function).
- The **bin** file dumper can be used by itself (via the **sys:dump-forms-to-file** function).
- The stream compiler can be used with the function compiler (**c-sh-C** or related Zmacs commands).
- All three compilers can be used (via **compile-file**, **compile-system**, or the Command Processor's **Compile System** command).

The following diagram shows the relationship of the different compilers to one another.



The Symbolics computer tools you use to invoke compilation determine the path through the diagram. For example, suppose you run the `compile-file` function on a Lisp source file. The function calls the stream compiler, which in turn calls the function compiler on any function definitions in the file. The function compiler passes the resulting compiled function objects to the `bin` file dumper. Some forms are passed directly to the `bin` file dumper (middle of the diagram) without being processed through the function compiler. All output from the `bin` file dumper is sent to a compiled code file. Loading that file creates the effect of compiling the source code directly to virtual memory.

For example, rather than compiling the source file, read it into an editor buffer and compile the entire buffer via the Zmacs command `Compile Buffer (n-X)`; the

output from the stream compiler and function compiler is evaluated immediately. The point is that while these two methods of compilation operate completely differently, the effect is the same once the results are in virtual memory.

## 12.1 How the Stream Compiler Handles Top-level Forms

The stream compiler accepts a stream of top-level Lisp forms and processes them. These forms are usually read from a stream of characters, which can be either a file or part or all of an editor buffer. The stream compiler categorizes these forms according to the table below and processes each according to its category. It calls the function compiler to translate a form that defines a function into a compiled function object containing compiled instructions. Certain other categories of forms are also processed specially, as documented in Table 1.

The stream compiler remembers certain "declarations" for the duration of the compilation. For example, when it compiles a macro definition, it saves the macro definition for use in processing subsequent top-level forms and function bodies. This permits a macro definition different from the one installed in the Symbolics computer's virtual memory to be used during compilation. Other kinds of "declarations" are also saved; most of these are documented in Table 1. The duration of the compilation during which these "declarations" are saved is usually a single invocation of the stream compiler, but when a system is being compiled (a program declared via `defsystem`) the declarations are in effect for the entire compilation, regardless of how many files in the system are compiled.

Stream compiler output can be sent either to the Symbolics computer virtual memory or to a file (via the `bin` file dumper) for later loading. This output can be regarded as a stream of forms that are evaluated either immediately, during the compilation, or later, when the `bin` file is loaded, depending on the type of compilation.

*Table 1. Lisp Forms that Require Special Processing by the Compiler.*

### 1. DEFINITIONS

Function Definitions, such as `(defun function-spec arguments body...)`, `(defselect...)`, and `(defmethod...)`

The stream compiler calls the function compiler to translate the function definition into a compiled function object. The result is to define the *function-spec* to be the compiled function object. See the function `fdefine` in *Symbolics Common Lisp: Language Dictionary*.

**Macro Definitions, such as (defmacro...)**

The stream compiler saves the definition of the macro for the duration of the compilation, and calls the function compiler to translate the function definition into a compiled function object. The result is to define the *function-spec* to be a macro whose expander function is the compiled function object. See the function **fdefine** in *Symbolics Common Lisp: Language Dictionary*.

**Substitutable Function Definitions, such as (defsubst...)**

The stream compiler saves the definition of the substitutable function for the duration of the compilation, and calls the function compiler to translate the function definition into a compiled function object. The result is to define the *function-spec* to be the compiled function object. See the function **fdefine** in *Symbolics Common Lisp: Language Dictionary*.

**Variable Definitions, such as (defvar...), (defparameter...), (zl:defconst...), (defconstant...), and (defvar-standard...)**

The stream compiler saves the declaration of the variable as a **special** variable for the duration of the compilation. It passes the form through as the compiler's output.

**Generalized Function Definitions: (def...) and (deff...)**

The stream compiler processes each subform of **def** after the initial function spec as a top-level form.

The stream compiler passes a **deff** form through as its output and remembers that it defines a function.

**Other Definitions, such as (defstruct...), (defflavor...), (defpackage...), and (defsystem...)**

The processing of each type of definition is idiosyncratic. The behavior of the stream compiler for these definition types is defined using the extension mechanisms discussed in this table, principally macro expansion.

**2. COMPILER-SPECIFIC FORMS****(progn form form...)**

Each *form* is processed as a top-level form. Any macro that expands into multiple top-level forms uses **progn** to arrange for the stream compiler to process all of the forms. See the section "Macros Expanding Into Many Forms" in *Symbolics Common Lisp: Language Concepts*.

**(eval-when (time time...) form form...)**

Each *form* is processed under the control of the list of *times*. If **load** is one of the *times*, the stream compiler processes each *form* as a top-level form. If **compile** is one of the *times*, each *form* is evaluated during the compilation.

**(compiler-let ((var val)...) form...)**

Each *form* is processed as a top-level form, with the specified bindings of **special** variables in effect.

**(function args...)** where the symbol *function* has a **compiler:top-level-form** property.

The value of the property must be a function of one argument. This function controls the behavior of the stream compiler.

### 3. DECLARATIONS

**(declare form form...)**

The stream compiler considers each *form*. If it invokes **special** or **unspecial**, the compiler handles it as if it had appeared at top level. Otherwise, the compiler simply evaluates *form*.

Use of **declare** in this way is considered to be an obsolete Maclisp-compatibility feature. Declaring special variables in a top-level **declare** form is not advisable because this hides the variables from interpreter, which uses **special** declarations in the same way as the compiler. It is preferable to declare **special** variables with an appropriate special form (such as **defvar**) that is understood by both the compiler and the interpreter, or by using **special** as a top-level form without enclosing it in **declare**, or by including a **(declare (special ...))** form inside the body of each function that uses the variable.

Forms to be evaluated at compile time should be specified with **eval-when** rather than **declare**. The stream compiler recognizes a top-level **(declare form1 form2...)** as equivalent to **(eval-when (compile) form1 form2...)** and evaluates *form1*, *form2*, and so on; if the car of *form* is **special** or **unspecial**, then that form is equivalent to **(eval-when (compile load) form)**. Forms appearing within a top-level **declare** should be valid top-level forms. Typical special forms that might appear are **special**, **unspecial**, **\*expr**, **\*lexpr**, and **\*fexpr**.

**(zl:local-declare (declaration declaration...) form form...)**

The stream compiler processes the *forms* as top-level forms, with the specified *declarations* in effect. **zl:local-declare** is considered to be an obsolete feature; use **declare** inside function bodies instead.

**(zl:special variable variable...)** and **(zl:unspecial variable variable...)**

The stream compiler saves the declaration for the duration of the compilation and outputs the form unchanged.



#### 4. OTHER FORMS

##### Macro Invocations

The stream compiler expands each top-level form that invokes a macro before further considering that form. Thus macro expansion can be used to extend the behavior of the stream compiler. Many definition forms are implemented by macros that expand into simpler definitions and other forms. For example, the expansion of such a macro might look like

```
(progn
  (record-source-file-name 'name 'type)
  (eval-when (compile)
    things to do at compile time)
  (defun ...))
```

For additional examples, use **mexp** to examine the expansion of **defvar**, **defsubst**, and **defstruct** forms.

##### Ordinary Forms

If the stream compiler does not recognize a form, it simply outputs the form unchanged.

##### Forms Protected From the Compiler

To prevent the stream compiler from recognizing a form, if for some reason it is necessary to pass the form unchanged through the compiler, the safest way is to conceal it inside an **eval** form. For example, the following form prevents the **foo** function from being converted into a compiled function object.

```
(eval (quote (defun foo (x) ...)))
```

##### Ignored Forms

The stream compiler ignores atoms (both variables and constants), (**quote x**), and (**zl:comment...**). It outputs no form when one of these appears in its input.

For Maclisp compatibility a number of top-level declaration forms are provided, including **zl:special**, **zl:unspecial**, **zl:\*expr**, **zl:\*lexpr**, and **zl:\*fexpr**.

---

##### **special** &rest *symbols*

*Special Form*

Declares each of the *symbols* to be "special" for the Lisp system (for example, the interpreter and the compiler). Provided for Maclisp compatibility. Note: **defvar** is usually preferred over **special**.

**zl:unspecial &rest symbols**

*Special Form*

Removes any "special" declarations of the *symbols* for the Lisp system (for example, the interpreter and the compiler). Provided for Maclisp compatibility.

### 12.1.1 Controlling the Evaluation of Top-level Forms

Sometimes you want to override the stream compiler's default behavior. For example, you might want a form to be put into the compiled code file (compiled, of course), or not; evaluated within the compiler, or not; or evaluated if the file is read directly into Lisp, or not. To tell the stream compiler exactly what to do with a form, use the general **eval-when** special form.

**eval-when** *times-list &body forms*

*Special Form*

**eval-when** allows you to tell the compiler exactly when the *body forms* should be evaluated. *times-list* can contain one or more of the symbols **load**, **compile**, or **eval**, or can be **nil**.

The interpreter evaluates the *body forms* only if the *times-list* contains the symbol **eval**; otherwise **eval-when** has no effect in the interpreter.

*If symbol is present*

*Then forms are*

**load**

Written into the compiled code file to be evaluated when the compiled code file is loaded, with the exception that **defun** forms put the compiled definition into the compiled code file.

**compile**

Evaluated in the compiler.

**eval**

Ignored by the compiler, but evaluated when read into the interpreter (because **eval-when** is defined as a special form there).

Example 1: Normally, top-level special forms such as **defprop** are evaluated at load time. If some macro expansion depends on the existence of some property, for example, *constant-value*, the definition of that property must be wrapped inside an (**eval-when** (**compile**) ...) so that the property is available at compile (macro expansion) time.

```
(eval-when (compile load eval)
  (defprop three 3 constant-value))
```

Example 2: **eval-when** should be used around **defconstants** of complex expressions. This is because the compiler does not maintain an environment acceptable to **eval** containing **defconstants**

```
(eval-when (compile load eval)
  (defconstant name expr))
```

In other words, if you are sure that (1) evaluating the *expr* in the global environment gives the correct results, and (2) that no harm is done by changing the current environment to have the (possibly new) value of *name*, then you can use the global environment as a substitute for the compilation environment.

In addition to **eval-when**, the **compiler:top-level-form** property provides another means for overriding the default behavior of the stream compiler.

### **compiler:top-level-form**

*Property*

The **compiler:top-level-form** property provides a way to extend the behavior of the stream compiler when it encounters a top-level form that looks like (*function* *args...*) and the symbol *function* has a **compiler:top-level-form** property. The value of the property must be a function of one argument. The compiler, rather than behaving in its normal fashion, calls the function with the original form as its argument. Whatever the function returns is dumped as the form to be evaluated at load time. You can have the function evaluate the form at compile time simply by calling **eval**. Note that the form returned by the function does *not* go back through the compiler's top-level form processing. This means that the returned form, which has been dumped to a compiled code file, cannot contain function definitions that you expect to be compiled.

## 12.2 Function Compiler

The function compiler takes a Lisp function and translates it from Lisp expressions into compiled functions. Compiled functions are represented in Lisp by compiled function objects, which contain machine code as well as various other information. The printed representation of the object is as follows:

```
#<DTP-COMPILED-FUNCTION name address>
```

When dealing with function bodies the function compiler performs the following operations on a form in this order:

1. Looks for compiler declarations (expands macros far enough to determine if they are declarations or not)
2. Performs style checking, unless you explicitly inhibit it.
3. Performs optimizations, if so requested, trying to optimize body forms from the inside out.
4. Runs transformations.
5. Expands macros.

If the case of a regular function, the entire process is repeated on the function's arguments. A special form, on the other hand, compiles its subforms, or not, depending on the syntax of the particular special form. When all the processing is done, the function compiler generates machine instructions.

## 12.3 Bin File Dumper

The **bin** (binary) file dumper accepts a stream of Lisp forms and/or machine-instruction function definitions from the function compiler and writes them in a compact form into a compiled code file.

It is also possible to make a compiled code file containing data, rather than a compiled program. Call the **bin** file dumper by itself via the **sys:dump-forms-to-file** function. See the section "Putting Data in Compiled Code Files", page 139.

By loading the compiled code file (using the function **load**, the Command Processor command **Load File**, or the Zmacs command **Load File [m-X]**) the objects represented in the file are created in your Lisp world.

## 12.4 Compiler Tools and Their Differences

### 12.4.1 Tools for Compiling Code From the Editor Into Your World

You can use several Zmacs commands to compile code in an editor buffer to your world. Users generally compile routines to memory as soon as they write them, debugging them before proceeding with more complex routines. The most common command for incremental compiling is **Compile Region (m-X)**, or **c-sh-C**.

**c-sh-C**

**Compile Region**

**Compile Region (m-X)**

Compiles the region, or if no region is defined, the current definition. Because recompiling routines as you edit them can be quite time-consuming, Zmacs provides two commands for compiling only those routines that have changed since they were last compiled: **Compile Changed Definitions (m-X)** and **Compile Changed Definitions of Buffer (m-X)**. These commands obviate the need to remember which routines have changed in your buffer or buffers. Alternatively, you can recompile the entire buffer.

**Compile Changed Definitions (m-X)**

Compiles any definitions that have changed in any of the current buffers. With a numeric argument, it prompts individually about whether to compile particular changed definitions (the default compiles all changed definitions).

**Compile Changed Definitions of Buffer (m-X)**

m-sh-C

Compiles any definitions that have changed in the current buffer. With a numeric argument, it prompts individually about whether to compile particular changed definitions. The default is to compile all changed definitions.

**Compile Buffer (m-X)**

Compiles the entire buffer. With a numeric argument, it compiles from point to the end of the buffer. (This is useful for resuming compilation after a prior Compile Buffer has failed.)

**12.4.2 Tools for Compiling Files**

Compiling a source file, using the Zmacs command **Compile File (m-X)**, the Command Processor command **Compile File**, or the function **cl:compile-file**, saves the output in a binary file (called a compiled code file). You can compile a file and also load the resulting file by using **compile-file** with the **:load** keyword set to **t**, or you can load the file separately into your Lisp world by using **load** or **Load File (m-X)**.

**Compile File Command**

Compile File *file-spec keywords*

Compile the file designated in *file-spec*.

<i>file-spec</i>	The pathname of the file to compile. The default is the usual file default.
<i>keywords</i>	:Compiler, :Load, :Query
:Compiler	{Lisp, use-canonical-type} The compiler to use. The default is use-canonical-type.
:Load	{yes, no, ask} Whether to load the file after compiling. The default is yes.
:Query	{yes, no, ask} Whether to ask for confirmation before compiling. The default is no.

**compile-file** *input-file* &key *output-file* *package* *load* *Function*  
 (*set-default-pathname*  
 \***compile-file-set-default-pathname**\*)

The file *input-file* is given to the compiler, and the output of the compiler is written to a file whose name is *input-file* with a canonical file type of **:bin**. *:output-file*, if supplied, lets you specify where the output is written. *:package* indicates the package with respect to which the *input-file* is compiled. If **t**, *:load* means to load the file after compiling it.

The purpose of **compile-file** is to take a file and produce a translated version that does the same thing as the original except that the functions are compiled. **compile-file** reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the compiled code file, which when loaded reproduces the effect of that source form.

Thus, if the source contains a (**defun** ...) form at top level, when the compiled code file is loaded, the function is defined as a compiled function. If, on the other hand, the source file contains a form that is not of a type known specially to the stream compiler, then that form (encoded in binary format) is output "directly" into the compiled code file, so that when that file is loaded that form is evaluated. For example, if the source file contains (**setq** *x* **3**), then the compiler places in the compiled code file instructions to set *x* to **3** at load time. (For a more general form, the compiled code file would contain instructions to recreate the list structure of a form and then call **eval** on it.)

**compile-file** returns the pathname of the *output-file*, which you can pass to **load** to load the compiled code file.

#### Compile File (m-k)

Compiles a file, offering to save it first (if it has an associated buffer that has been modified). It prompts for a file name in the minibuffer, using the file associated with the current buffer as the default. It does not load the file.

#### 12.4.2.1 File Types of Lisp Source and Compiled Code Files

The results of compilation are written to a file of canonical type **:bin**. The actual file types for compiled code files are host-dependent, as are those of the Lisp source files. The following table shows the file types of both input and output files for various hosts.

<i>Host type</i>	<i>File type of source file</i>	<i>File type of compiled code file</i>
Symbolics computer	<b>lisp</b>	<b>bin</b>

Multics	lisp	bin
TOPS-20	LISP, LSP	BIN
UNIX	l, lisp	bn, bin
VAX/VMS	LSP	BIN

### 12.4.3 Tools for Compiling Single Functions

Compiled functions are Lisp objects that contain programs in the machine instruction set. Compiling an interpreted function by calling the function compiler on a function spec, converts it into a compiled function and changes the definition of the function spec to be that compiled function. Most users do not compile functions directly, but rather compile files or regions of code in a Zmacs buffer.

**compile** *function-spec* &optional *lambda-exp* *Function*

**compile** gets the function definition from either of its arguments. If the lambda expression *lambda-exp* is supplied, **compile** uses *lambda-exp* and converts it into a compiled function object. If, on the other hand, *lambda-exp* is **nil**, **compile** gets the function definition of *function-spec*, which is either a function specification or **nil**. If **nil**, **compile** returns the compiled function object without storing it anywhere. If *function-spec* is not **nil**, **compile** changes *function-spec*'s definition to be the compiled function object; the returned value is *function-spec*.

See the function **fdefine** in *Symbolics Common Lisp: Language Dictionary*.

**uncompile** *function-spec* *Function*

If *function-spec* is not defined as an interpreted function and it has a **:previous-expr-definition** property, then **uncompile** restores the function cell from the value of the property. (Otherwise, **uncompile** does nothing and returns "Not compiled".) This "undoes" the effect of **compile**. See the function **undefun** in *Symbolics Common Lisp: Language Dictionary*.

Although all these methods call the compiler and produce compiled function objects, they are not equivalent. For example, using **compile-file** to compile a source file of canonical type **:lisp** converts it into a binary file, with a canonical file type of **:bin**. Compiling the source file has no effect on your Lisp environment. Compiling a top-level form in an editor buffer, using a command like Compile Region (c-sh-C) or Compile Buffer (m-X), creates a compiled function object in memory but does not write an object code file on disk. Compiling a top-level form in an editor buffer does cause some side effects on the Lisp environment.

The most essential difference, however, between compiling a source file and compiling the same code in an editor buffer is this: When you compile a file, most function specs are not defined and most forms (except those within **eval-when** (**compile**) forms) are not evaluated at compile time. Instead the

compiler puts instructions into the binary file that causes evaluation to occur at load time.

Loading a compiled code file does not differ substantially from loading its associated source file, except that the functions defined in the binary file are defined as compiled functions instead of interpreted functions. When you load a source file that contains `defun` forms, you define the function specs named in the forms to be those functions.

Sometimes you might want to put things in the compiled code file that are not meant merely to be translated into binary form. Top-level macro definitions fall into this category. The macros must actually get defined within the compiler in order for the compiler to be able to expand them at compile time. Compiler declarations also fall into this category.





## 13. Compiler Warnings Database

Compiler warnings are kept in an internal database. Several functions, Command Processor commands, and Zmacs commands allow you to inspect and manipulate this database in various ways.

The database of compiler warnings is organized by pathname; warnings that were generated during the compilation of a particular file are kept together, and this body of warnings is identified by the generic pathname of the file being compiled. Any warnings that were generated while compiling some function not in any file (for example, by using the **compile** function on some interpreted code) are stored under the pathname **nil**. For each pathname, the database has entries, each of which associates the name of a function (or a flavor) with the warnings generated during its compilation.

The database starts out empty when you cold boot. Whenever you compile a file, buffer, or function, the warnings generated during its compilation are entered into the database. If you recompile a function, the old warnings are removed, and any new warnings are inserted. If you get some warnings, fix the mistakes, and recompile everything, the database becomes empty again.

Warnings can also be saved to a file or printed out as well as stored in the database. If the value of the special variable **compiler:suppress-compiler-warnings** is not **nil**, warnings are not printed, although they are still stored in the database.

### Save Compiler Warnings Command

Save Compiler Warnings *pathname files-whose-warnings-to-save*

Save compiler warnings of the files *files-whose-warnings-to-save* to the specified *pathname*. *files-whose-warnings-to-save* can be **All** to save all warnings, or it can be a list of one or more pathnames. Among the pathnames can be the special token **No File** to catch warnings for no particular file.

The database has a printed representation. The command **Show Compiler Warnings** or the function **print-compiler-warnings** produces this printed representation from the database, and **compiler:load-compile-warnings** updates the database from a saved printed representation.

### Show Compiler Warnings Command

Show Compiler Warnings *pathname(s-or-special-tokens) keyword*

Display compiler warnings of the files specified by *pathnames* or use the special

tokens All (to show all compiler warnings) or No File to show the warnings for no particular file. The only valid keyword is Output Destination, which is a stream to which to direct the output.

**print-compiler-warnings** &optional *files* (*stream* **zl:standard-output**) *Function*  
*file-node-message* *function-node-message*  
*anonymous-function-node-message*

Prints out the compiler warnings database. If *files* is **nil** (the default), it prints the entire database. Otherwise, *files* should be a list of generic pathnames, and only the warnings for the specified files are printed. (**nil** can be a member of the list, too, in which case warnings for functions not associated with any file are also printed.) The output is sent to *stream*, which you can use to send the results to a file.

**compiler:load-compiler-warnings** *file* &optional (*flush-old-warnings* *t*) *Function*

Updates the compiler warnings database. *file* should be the pathname of a file containing the printed representation of the compiler warnings related to the compilation of one or more files. If *flush-old-warnings* is **t** (the default), any existing warnings in the database for the files in question are completely replaced by the warnings in *file*. If *flush-old-warnings* is **nil**, the warnings in *file* are added to those already in the database.

The printed representation of a set of compiler warnings is sometimes stored in a file. You can create such a file using **print-compiler-warnings**, but it is usually created by invoking **compile-system** with the **:batch** option. The default type for such files is CWARNS. For example, FOO.CWARNS.

Several Zmacs commands manipulate the compiler warnings database.  
 Compiler Warnings (m-X)

Creates the compiler warnings buffer (called \*Compiler-Warnings-1\*) if it does not exist, puts all outstanding compiler warnings in that buffer, and switches to that buffer. You can view the compiler warnings by scrolling around and doing text searches through them using Edit Compiler Warnings (m-X).

Edit Compiler Warnings (m-X)

Prompts you with the name of each file mentioned in the database, allowing you to edit the warnings for that file. It then splits the Zmacs frame into two windows: the upper window displays a warning message and the lower one displays the source code whose compilation caused the warning. After you have finished editing each function, **c-.** gets you to the next warning: the top window scrolls to show the next warning and the bottom window displays the function associated with this warning. Successive **c-.**s take you through all of the warning messages for all of the files you specified. When you are done, the last **c-.** puts the frame back into its previous configuration.

**Edit File Warnings (m-X)**

Asks you for the name of the file whose warnings you want to edit. You can give either the source file or the compiled file. Only warnings for this file are edited. If the database does not have any entries for the file you specify, the command prompts you for the name of a file that contains the warnings, in case you know that the warnings are stored in another file.

**Load Compiler Warnings (m-X)**

Loads a file containing compiler warning messages into the warnings database. It prompts for the name of a file that contains the printed representation of compiler warnings. It always replaces any warnings already in the database.



## 14. Controlling Compiler Warnings

### 14.1 Compiler Style Warnings

The compiler performs style checking on all forms. This means that the Lisp compiler produces compiler warnings when it sees programs that are invalid Lisp or that may produce errors at runtime. You can add to the checks that the compiler makes in several ways.

- Your macros can call the `warn` function to warn of problematic usage.
- You can use `compiler:make-obsolete` to declare something obsolete.
- You can define *style checkers* by means of the function-spec `compiler:style-checker`. A style checker is a Lisp function associated with a symbol. When the compiler compiles an s-expression with that symbol in the functional position `car`, it calls all of the style checkers for the symbol with an argument of the form. These style checkers can examine the form and call `warn` if they detect something wrong.

`compiler:style-checker` *checker-name symbol &optional form* *Function*

Define a style checker. Note: `compiler:style-checker` is not a function but rather, a function-spec. A style checker is a Lisp function associated with a symbol. When the compiler compiles an s-expression with that symbol in the functional position `car`, it calls all of the style checkers for the symbol with an argument of the *form*. These style checkers can examine the *form* and call `warn` if they detect something wrong. *checker-name* is the name of your style checker function, and *symbol* is the symbol that you want to check. *arg1* and *arg2* are optional arguments to your style checker function. For example:

```
(compiler:style-checker fs:obsolete-arguments open)
```

detects old unsupported calls to `open` at compile time.

You define a style checker as follows:

```
(defun (compiler:style-checker style-checker-name function-symbol)
  (form)
  ... body that looks at the form ...
)
```

You can have multiple style checkers on a single function symbol. For example, assume that you define function to take a first argument that must be a number, and which is often a constant.

```
(defun stylish-function (number &rest other-args)
  )
```

You might write:

```
{defun (compiler:style-checker first-arg-must-be-numeric stylish-function) (form)
  (destructuring-bind (ignore number &rest ignore) form
    (when (and (compiler:constant-form-p number)
              (not (numberp (compiler:constant-evaluator number))))
      (warn "The first argument ~S to ~S is not a number." number 'stylish-function))))
```

In the example, the function **compiler:constant-form-p** simply checks if the form is treated as a constant by the compiler; the function **compiler:constant-evaluator** returns the value of a constant. You have to be very careful about how you examine arguments. The **form** in the example code is un-compiled list structure. If the caller is passing a variable as an argument

```
(stylish-function foo)
```

then the form will contain the symbol **foo** as the second element. **foo** is not a constant, so you cannot tell what its runtime value is at compile time.

The pre-Genera 7.0 way of style checking using property lists is also supported, but you cannot use both the new and the old technology on the same checked function. In the old way, style checking is implemented by the **compiler:style-checker** property on a symbol; the value of the property is called on all forms whose **car** is that symbol, except those immediately enclosed in **inhibit-style-warnings**. Obsolete function warnings are also performed by means of the style-checking mechanism.

**inhibit-style-warnings** *form*

*Macro*

Prevents the compiler from performing style-checking on the top level of *form*; style-checking will still be done on the arguments of *form*.

The following code warns you about the obsolete function **zl:explode**, since **inhibit-style-warnings** applies only to the top level of the form inside it, in this case, to the **setq**.

*Generate warning:*

```
(inhibit-style-warnings (setq bar (explode foo)))
```

The following code, on the other hand, does *not* warn that **explode** is an obsolete function:

*Do not generate warning:*

```
(setq bar (inhibit-style-warnings (explode foo)))
```

If an optimizer needs to return a form with nested "bad-style" forms, there should be an explicit **inhibit-style-warnings** wrapped around the nested forms.

By setting the compile-time value of **inhibit-style-warning-switch** you can enable or disable some of the warning messages of the compiler. The compile-time value of **obsolete-function-warning-switch** enables or disables obsolete-function warnings in particular.

**compiler:make-obsolete** *spec reason* &optional (*type 'type-arg*) *Special Form*

**compiler:make-obsolete** is a special form that declares a function, flavor, or structure to be obsolete; code that calls an obsolete definition generates a compiler warning. It is useful for marking as obsolete some Maclisp functions that exist in Zetalisp but should not be used in new programs, or for reminding users that some function is being phased out.

*spec* is the definition to be made obsolete and is not evaluated. *reason* is evaluated and is the warning or explanation to be printed when the obsolete definition is called. *type-arg*, the optional third argument, is the definition-type of the object declared obsolete and is not evaluated. Its default value is **defun** when no type is specified. **compiler:make-obsolete** recognizes four definition-types: **defun**, **defflavor**, **defstruct**, and **defvar**.

**compiler:make-obsolete** with a third argument of **defstruct** makes the structure obsolete as well as all of its accessor functions.

**compiler:make-obsolete** with a third argument of **defflavor** makes obsolete both the flavor and its outside accessible instance variables.

An attempt to create a new flavor with an obsolete flavor as an included or component flavor generates a compiler warning. Likewise, creating a new structure with an obsolete structure as an included structure also generates a warning.

**compiler:make-message-obsolete** *message-name format-string* *Special Form*

Allows you to generate compiler warnings about obsolete message names. The first argument, *message-name*, is the obsolete message name. The second argument, *format-string*, is the warning to be printed. If the string contains the `~S` format directive, it will be replaced by the object that was sent the message.

Example:

```
(compiler:make-message-obsolete :clear-screen
  "You have sent the message :CLEAR-SCREEN to the object ~S.
  This name is obsolete. The new name for this message is
  :CLEAR-WINDOW. Please update your code.")
```



## 14.2 Function-referenced-but-never-defined Warnings

Normally, the compiler notices whenever any function *x* calls any other function *y*; it takes note of all these uses, and then warns you at the end of the compilation if function *y* was called but was neither defined nor declared (by **compiler:function-defined**).

The compiler uses a set of variables and functions to keep track of which functions have been defined and which have been referenced. These are the basis for the messages "FOO was defined but never referenced" that occur during compiling.

**sys:file-local-declarations** *Variable*

**sys:file-local-declarations** stores global declarations valid for the entire compilation. Since it can become fairly large, it is implemented as a hash table (or nil). The symbol being declared is the key, and the value is a property list of declarations and values. The default value is nil.

**compiler:functions-defined** *Variable*

**compiler:functions-defined** is a hash table of all functions defined or nil, if none has been defined yet.

**compiler:functions-referenced** *Variable*

**compiler:functions-referenced** is a hash table of functions referenced but not defined. Each entry is an alist of (<generic-pathname> . <by-whom>). In this way warnings can be put into the appropriate file when this variable is processed at the end of a compilation.

**compiler:function-defined *fspec*** *Function*

**compiler:function-defined** tells the compiler that the function *fspec* has been defined (by putting it into the hash table in **compiler:functions-defined**).

**zl:\*expr**, **zl:\*lexpr**, and **zl:\*fexpr** are the Maclisp equivalents of **compiler:function-defined**.

**zl:\*expr &rest *functions*** *Special Form*

Declares each function spec in the list of *functions* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, which appears at the end of the compilation. Provided for Maclisp compatibility.

**zl:\*lexpr &rest *functions*** *Special Form*

Declares each function spec in the list of *functions* to be the name of a function. In addition it prevents these functions from appearing in the list

of functions referenced but not defined that is printed at the end of the compilation. Provided for Maclisp compatibility.

**zl:\*fexpr** &rest *functions* *Special Form*

Declares each function spec in the list of *functions* to be the name of a special form. In addition it prevents these names from appearing in the list of functions referenced but not defined that is printed at the end of the compilation. Provided for Maclisp compatibility.

**compiler:file-declare** *thing declaration value* *Function*

**compiler:file-declare** enters a declaration in the table **sys:file-local-declarations** for the remaining extent of the compilation environment.

```
(compiler:file-declare 'foo 'special t)
```

**compiler:file-declaration** *thing declaration* *Function*

**compiler:file-declaration** looks up a declaration in the table **sys:file-local-declarations**. It returns the declaration when *thing* is a declaration of type *declaration* and **nil** otherwise.

**compiler:function-referenced** *what* &optional (*by* *Function*

**compiler:default-warning-function**)

**compiler:function-referenced** is useful for requesting compiler warnings in certain esoteric cases. For example, sometimes the compiler has no way of telling that a certain function is being used. Suppose that instead of *x*'s containing any forms that call *y*, *x* simply stores *y* away in a data structure somewhere, and someplace else in the program that data structure is accessed and **funcall** is done on it. In this case the compiler cannot see that this is going to happen; the result is that it cannot note the function usage and hence cannot create a warning message. In order to make such warnings happen, you can explicitly call the function **compiler:function-referenced** at compile-time.

*what* is a symbol that is being used as a function. *by* can be any function spec. **compiler:function-referenced** must be called at compile time while a compilation is in progress. It tells the compiler that the function *what* is referenced by *by*. When the compilation is finished, if the function *what* has not been defined, the compiler issues a warning to the effect that *by* referred to the function *what*, which was never defined.

#### 14.2.1 Overriding Variable-defined-but-never-referenced Warnings

Sometimes functions take arguments that they deliberately do not use. Normally the compiler warns you if your program binds a variable that it never references. In order to disable this warning for variables that you know you are not going to use, you can do one of several things.

- You can declare the variable to be ignored:

```
(declare (ignore fraz-size))
```

- You can name the variables **ignore** or **ignored**. The compiler does not complain if a variable of one of these names is not used. Furthermore, you can have more than one variable in a lambda-list that has one of these names.
- You can simply use the variable for effect (ignoring its value) at the front of the function. This has the advantage that **arglist** will return a more meaningful argument list for the function, rather than returning something with **ignores** in it. Example:

```
(defun the-function (list fraz-name fraz-size)
  fraz-size      ; This argument is not used.
  ...)
```

- You can use the variable as an argument to the **ignore** function.

```
(defun the-function (list fraz-name fraz-size)
  (ignore fraz-size)
  ...)
```

## 15. Compiler Switches

The compile-time values of the following variables, so-called "compiler switches", affect the operation of the compiler. Use `compiler-let` to bind compiler switches.

**compiler:obsolete-function-warning-switch** *Variable*

The compile-time value of this variable affects the operation of the compiler. If this variable is non-`nil`, the compiler tries to warn you whenever an obsolete function, such as `zl:maknam` or `zl:samepnamep`, is used. The default value is `t`.

**compiler:open-code-map-switch** *Variable*

The compile-time value of this variable affects the operation of the compiler. If this variable is non-`nil`, the compiler attempts to produce inline code for the mapping functions (`mapc`, `mapcar`, and so on, but not `zl:mapatoms`) if the function being mapped is an anonymous lambda-expression. Setting this switch to `nil` makes the compiled code smaller. Setting this switch to `t` makes the compiled code larger but faster. The default value is `t`.

**zl:all-special-switch** *Variable*

The compile-time values of this variable affects the operation of the compiler. If this variable is non-`nil`, the compiler regards all variables as special, regardless of how they were declared. The default is `nil`.

**compiler:inhibit-style-warnings-switch** *Variable*

The compile-time values of this variable affects the operation of the compiler. If this variable is non-`nil`, all compiler style-checking is turned off. Style checking is used to issue obsolete function warnings and other sorts of warnings. The default value is `nil`.

**compiler:compiler-verbose** *Variable*

The compile-time values of this variable affects the operation of the compiler. The compiler displays a message (using `zl:standard-output`) each time it starts compiling a function when the value of `compiler:compiler-verbose` is `t`. The default value is `nil`.



## 16. Compiler Source-Level Optimizers

An *optimizer* is a function that converts a form into another form that is more efficiently executed. An optimizer can be used to transform code into an equivalent but more efficient form that can be compiled better. For example, `(eq obj nil)` is transformed into `(null obj)`, which can be compiled better.

Do not use optimizers to define new language features, because they take effect only in the compiler; the interpreter (that is, the evaluator) does not know about optimizers. So an optimizer should not change the effect of a form; it should produce another form that does the same thing, possibly faster or with less memory. If you want to actually change the form to do something else, you should use macros.

The compiler treats (optimized or transformed) forms returned by compiler optimizers as if they were wrapped in an `inhibit-style-warnings` form. For example, the expression:

```
(eq 1 × 3)
```

is optimized into the expression:

```
(eq × 3)
```

In general, it is a bad idea to compare numbers with `eq`, since the implementation of numbers is such that some numbers can be compared with `eq` and some can't. A style checker keeps the user from writing `(eq × 3)`. The optimizer is allowed to do this without warning on the assumption that the optimizer always generates "correct" code.

Note: `inhibit-style-warnings` only affects the top-level form inside it. If an optimizer needs to return a form with nested "bad-style" forms, there should be an explicit `inhibit-style-warnings` wrapped around the nested forms.

**compiler:add-optimizer** *target-function optimizer-name &rest* *Special Form*  
*optimized-into*

Puts *optimizer-name* on *target-function*'s optimizers list if it is not there already. *optimizer-name* is the name of an optimization function, and *target-function* is the name of the function calls that are to be processed. Neither is evaluated.

(**compiler:add-optimizer** *target-function optimizer-name optimize-into-1 optimize-into-2...*) also remembers *optimize-into-1*, and so on, as names of functions that can be called in place of *target-function* as a result of the optimization.



## 17. Files That Maclisp Must Compile

Certain programs are intended to be run both in Maclisp and in Symbolics-Lisp. Their source files need some special conventions. For example, all **special** declarations must be enclosed in top-level **declare** forms, so that the Maclisp compiler sees them. The main issue is that many Symbolics-Lisp functions and special forms do not exist in Maclisp.

The "#q" sharp-sign reader macro causes the object that follows it to be visible only when compiling for Symbolics-Lisp. The sharp-sign reader macro #m causes the following object to be visible only when compiling for Maclisp. These work both on subexpressions of the objects in the file, and at top level in the file. To conditionalize top-level objects, however, it is better to put the macros **zl:if-for-lispm** and **zl:if-for-maclisp** around them. (You can only put these around a single object.) The #q sharp-sign reader macro cannot do this, since it can be used to conditionalize any Lisp object, not just a top-level form.

To allow a file to detect what environment it is being compiled in, the following macros are provided:

**zl:if-for-lispm** &rest *forms* *Macro*

Seen at the top level of the compiler, *forms* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Symbolics-Lisp. If the Symbolics-Lisp interpreter sees this it evaluates *forms* (the macro expands into *forms*).

**zl:if-for-maclisp** &rest *forms* *Macro*

Seen at the top level of the compiler, *forms* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Maclisp (for example, if the compiler is COMPLR). If the Symbolics-Lisp interpreter sees this it ignores it (the macro expands into **nil**).

**zl:if-for-maclisp-else-lispm** *maclisp-form lispm-form* *Macro*

When (**if-for-maclisp-else-lispm** *form1 form2*) is seen at the top level of the compiler, *form1* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Maclisp; otherwise *form2* is passed to the compiler top level.

**zl:if-in-lispm** &rest *forms* *Macro*

In Symbolics-Lisp, (**if-in-lispm** *forms*) causes *forms* to be evaluated; in Maclisp, *forms* is ignored.



**zl:if-in-maclisp** &rest *forms**Macro*

In Maclisp, (**if-in-maclisp** *forms*) causes *forms* to be evaluated; in Symbolics-Lisp, *forms* is ignored.

When you have two definitions of one function, one conditionalized for one machine and one for the other, put them next to each other in the source file with the second "(defun)" indented by one space, and the editor will put both function definitions on the screen when you ask to edit that function.

In order to make sure that those macros are defined when reading the file into the Maclisp compiler, you must make sure the file starts with a prelude, which should look like:

```
(declare (cond ((not (status feature lispm))
                (load '|AI: LISPM2; CONDIT|))))
```

This does nothing when you compile the program on Symbolics computers. If you compile it with the Maclisp compiler, it loads definitions of the above macros, so that they will be available to your program. The form (**status feature lispm**) is generally useful in other ways; it evaluates to **t** when evaluated on Symbolics computers and to **nil** when evaluated in Maclisp.

## 18. Putting Data in Compiled Code Files

A compiled code file can contain data rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in a printed representation of that same data structure. Also, certain data structures, such as arrays, do not have a convenient printed representation as text, but can be saved in compiled code files.

In compiled programs, the constants are saved in the compiled code file in this way. The compiler optimizes by making constants that are `zl:equal` become `eq` when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a compiled code file is loaded, objects that were `eq` when the file was written are still `eq`; this does not normally happen with text files.

The following types of objects can be represented in compiled code files:

- Symbols
- Numbers of all kinds
- Lists
- Strings
- Arrays of all kinds
- Instances (for example, hash tables)
- Compiled function objects

When an instance is put (dumped) into a compiled code file, it is sent a `:fasd-form` message, which must return a Lisp form that, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply dumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain `eq`; the `:fasd-form` message is sent only the first time a particular instance is encountered during writing of a compiled code file. If the instance does not accept the `:fasd-form` message, it cannot be dumped.

**sys:dump-forms-to-file** *filename forms &optional file-attribute-list* *Function*  
**sys:dump-forms-to-file** writes data to a file in binary form. *forms-list* is a list of Lisp forms, each of which is dumped in sequence. It dumps the forms, not their results. The forms are evaluated when you load the file.

For example, suppose `a` is a variable bound to any Lisp object, such as a list or array. The following example creates a compiled code file that recreates the variable `a` with the same value:

```
(sys:dump-forms-to-file "f:>foo>aval"
  (list '(setq a ',a)))
```

For the purposes of understanding what this function does, you can consider that it is the same as the following:

```
(defun sys:dump-forms-to-file (file forms)
  (with-open-file (s file ':direction ':output)
    (dolist (f forms)
      (print f s))))
```

The actual definition (which is more complicated) writes a binary file in a more easily parsed format so it will load faster. It can also dump arrays, which you cannot write to a Lisp source file.

*attribute-list* supplies an optional attribute list for the resulting compiled code file. It has basically the same result when loading the binary file as the file attribute list does for **compiler:compile-file**. Its most important application is for controlling the package that the file is loaded into.

```
(sys:dump-forms-to-file "foo" forms-list '(:package "user"))
```

**sys:dump-forms-to-file** always puts a package attribute into the binary file it writes. If you do not specify the *attribute-list* argument, or if *attribute-list* does not contain a **:package** attribute, the function uses the **cl-user** or **zl-user** package, depending on the context. This is to ensure that package prefixes on symbols are always interpreted when they are loaded as they were intended when the file was dumped.

The *file-attribute-list* argument can be used to store useful information (such as "headers" for special data structures) in the file's attribute list. The information can then be retrieved from the attribute list with **fs:pathname-attribute-list**, without reading the rest of the file.

## **PART IV.**

### **Maintaining Large Programs**



## 19. Introduction to the System Construction Tool

### The Need to Maintain Programs as Systems of Files

When a program becomes large, it is often desirable to split it up into several files. One reason is to help keep the parts of the program organized, to make things easier to find. Another is that programs broken into small pieces are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any piece of it changes; if the program is broken up into many files, only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more mechanism is needed to manipulate it. To load the program, you now have to load several files separately, instead of just loading one file. To compile it, you have to figure out which files need compilation, by seeing which have been edited since they were last compiled, and then you have to compile those files.

An even more complicated factor is that files can have interdependencies. You might have a Lisp file called "defs" that contains macro definitions (or flavor definitions), and functions in other files might use those macros. This means that in order to compile any of those other files, you must first load the file "defs" into the Lisp environment, so that the macros will be defined and can be expanded at compile time. You would have to remember this whenever you compile any of those files. Furthermore, if "defs" has changed, other files of the program might need to be recompiled because the macros might have changed and need to be reexpanded.

Finally, you might want to generate multiple versions of the program – a stable version for general users to run, another for development purposes; source control for the various versions would be nearly impossible to maintain manually.

### The Symbolics System Construction Tool

This chapter describes the *System Construction Tool* (SCT), which addresses these difficulties. A *system* is a set of files and a set of rules and procedures that define the relations among these files; together these files, rules, and procedures constitute a complete program.

SCT examines the creation times of the source files to determine which ones must be recompiled to produce "clean" and "coherent" object files. It can also be used to merge patches to a system. By tracking all dependencies, SCT ensures that each released system is consistent. See the section "Directories Associated with a System", page 191.

A system can be constructed out of Lisp source files or files written in other languages. Systems can also be constructed out of text files (for example, the documentation system) or other types of files specified by users.

- You define the system, using SCT's **defsystem** special form. The definition, called a *system declaration*, specifies such information as the names of the source files (or modules) in your system and what operations should be performed on each file in what order (for example, which files should be compiled, loaded, or both, and which should be loaded first). See the section "Defining a System", page 145.
- The body of a **defsystem** declaration names the files that compose the system and consists of one or more *module* specifications. A module is one or more files or modules that should be treated as a unit. *Operations* – compiling, loading, editing, hardcopying, and the like – are applied to the module as a whole. See the section "**defsystem** Modules", page 153.
- If the system is to be made generally available to other users, you should place the system definition in its own file. (This file should contain no more than one **defsystem** form, but there can be any number of **defsubsystems** and other forms.) You also must create two other files that make your system site-independent. The goal is to make your system run at any site, not just the one on which it physically resides. (Imagine the problems that would occur if you moved your program to another host machine, and you had to update every single pathname listed in your system definition!)
- You can perform *operations* on your system (for example, compile, edit, load, reap-protect, distribute, release, or hardcopy) by using the appropriate Command Processor commands (e.g., Load System and Compile System) or Lisp functions. See the section "Loading and Compiling Systems", page 175. See the section "Other Operations on Systems", page 187. You can also define your own operations to perform on systems. See the section "User-defined Operations on Systems", page 172.
- The patch facility lets you make and distribute incremental fixes and improvements to your system, called *patches*, thereby avoiding recompilation or reloading of the entire system. By maintaining a patch registry, a detailed record keeping system, the patch facility allows developers to maintain multiple versions of the same system. See the section "Patch Facility", page 197.
- Various functions exist to help you find information about existing systems. See the section "Obtaining Information About a System", page 213.

## 20. Defining a System

A *system* is a set of files and a set of rules and procedures that defines the relations among these files; together these files, rules, and procedures constitute a complete program. The definition of a system (called the *system declaration*) describes these relationships and rules. Some useful, general guidelines are:

1. Use Zmacs to enter the system declaration in its own file, with a canonical type of `:lisp`. The system declaration file also contains a package declaration for the system (if necessary), which must precede the system declaration in the file. For an example of a system declaration file: See the section "System Declaration File", page 183.
2. Create a `defsystem` form. Wherever a pathname is required in your system declaration use logical pathnames, not physical pathnames. Logical pathnames provide a way of referring to files in a site-independent way. They also make it possible to move the sources from one machine to another within a site.
3. Assuming that you have used logical pathnames, you need to prepare two other files:
  - The *system* file
  - The *translations* file

The system file defines a logical host, specifies the location of the system declaration file, and loads the translations file. The translations file defines the translation from logical directories on the logical host to physical directories on a physical host. See the section "Loading System Definitions That Use Logical Pathnames", page 180.

4. Invoke a Lisp function or Command Processor (CP) command to compile, load, or perform some other operation on your system, as in

```
Load System Fortran :Version Latest
```

The command uses the information in the translations file to load the system declaration file, compiling this declaration file first if necessary.



**defsystem** *system-name options &body body* *Special Form*

Defines a system called *system-name*. This name is used for all operations on the system.

The definition of a system (called the *system declaration*) describes a group of relations among a group of files that constitute at least one complete program. The declaration provides information on (1) the files that make up the system, (2) which files depend on previous operations, and (3) the characteristics of the system, for example, the package in which the source code should be read. Note: *system-name* is not package-dependent. It is only used as a string.

Interpreting or compiling the system declaration brings your system into existence for the purposes of applying operations to it. After your system declaration is loaded into the Lisp environment, Command Processor commands (like Load System and Compile System) and corresponding Lisp functions construct a plan of operations in accordance with the properties specified in your system declaration. The system is operated on according to this plan.

*options* is a list of keyword and value pairs that specify global attributes of the system being defined. *body* contains the detailed specification of the parts of the system. *body* can be written using a *long-form syntax* or an abbreviated *short-form syntax*.

### 20.0.1 defsystem Options

*options* is a list of keyword and value pairs that specify global attributes of the system being defined.

**:pretty-name**     **:pretty-name** specifies the name of the system for use in printing. This is the user-visible name appearing in heralds and so on. If **:pretty-name** is not specified, the default is the name of the system: SYSTEM NAME.

Example: Based on the following declaration, the herald displays the name of the registrar system as **Automatic Registration System**.

```
(defsystem registrar
  (:pretty-name "Automatic Registration System"
   :short-name "Registration"
   :default-pathname "reg:reg;")...)
```

**:short-name**     **:short-name** specifies an abbreviated name used in constructing disk label comments and patch file names for some file systems. See the section "Names of Patch

Files", page 201. If the **:short-name** is not supplied, *system-name* is used.

### **:default-package**

**:default-package** specifies the name of an existing package into which each file in the system will be loaded or compiled. This is only useful if the file has no package attribute in its mode line. (Typically, the package declaration for a system is placed in the same file as the system declaration.) (See the section "Defining a Package" in *Symbolics Common Lisp: Language Concepts*.)

It is sometimes necessary to selectively override the system's default-package, for example, when a particular system module needs to read a file into a particular package. In this case specify a different package for a particular module. See the section "**:module** Keyword Options", page 159.

On other occasions you might want to compile or load your system in a package other than the default package for purposes of debugging new versions of the system. See the section "**defsystem** Options", page 146.

Example: All the modules in **mailer**, except for **macros**, are compiled/loaded into the **mail** package.

```
(defpackage mail (:size 4096.))
```

```
(defsystem mailer
  (...
   :default-package mail
  ...))
(:module defs "defs")
(:module macros "macros" (:package special)
  (:in-order-to :compile (:load defs)))
...)
```

Note: Your system should be compiled and loaded in its own unique package. If your system and someone else's system both define a function called **foo**, but with different package names, the package specification will prevent name conflicts. Avoid affecting symbols in the standard Genera packages. See the section "Packages" in *Symbolics Common Lisp: Language Concepts*.

**:package-override**

The **:package-override** option overrides all other explicit package declarations – the system default package, a package declaration for a particular module, as well as a package specified in the attribute lists of constituent files.

Commonly, this option is used when debugging a new version of a system. For example, temporarily insert the option in your **defsystem** form, reevaluate the form, and compile and test your experimental version. Do not save the system declaration file with the **:package-override** option. When you're finished debugging the new version, delete the option from the **defsystem** form and reevaluate it.

**:default-pathname**

**:default-pathname** specifies a default pathname against which all other pathnames in the system are merged. Specify that part of the pathname for which you want to establish a default. You are urged to supply a logical, not a physical, pathname. See the section "Logical Pathnames" in *Reference Guide to Streams, Files, and I/O*.

Here is an example.

```
:default-pathname "sys:zwei;"
```

This eliminates the need to enter the full pathname of each of the system's files. If the system's files reside in more than one directory, furnish a pathname default for the directory storing the largest number of files. Where the pathname differs from the default, specify the full pathname.

Example: "pres-type-macro" and "pres-type-fspec" are merged here into "sys:dyno-windows;pres-type-macro" and "sys:dyno-windows;pres-type-fspec" respectively. Because "character-style-pres" resides in "sys:sys2;" a full pathname specification is given.

```
(defsystem dyno-windows
  (:pretty-name "Dynamic Windows"
   :default-pathname "sys:dyno-windows;")
  (:serial (:parallel "pres-type-macro" "pres-type-fspec")
           (:parallel "sys:sys2;character-style-pres"))))
```

**:default-module-type**

Specifies a keyword, which is the default type for each

module. If not furnished, the default value is `:lisp`. The type specifies the nature of the inputs to the system and determines the details of what is done for each generic operation (load, edit, hardcopy) performed on the system.

Some commonly used predefined types are: `:lisp`, `:fortran`, `:pascal`, `:text`, `:font`, `:lisp-example`, and `:system`. (The `:fortran` and `:pascal` types are supplied by the corresponding optional products.) For a complete list of predefined types and operations: See the section "Table of Module Types and Operations", page 169.

You can also define your own types. See the section "User-defined Module Types", page 171.

It is possible to selectively override the system's default type by specifying another type for a particular module. See the section " :module Keyword Options", page 159.

Example: The `action` system consists of one anonymous (unnamed) module of type `:fortran`.

```
(defsystem action
  (:short-name "act"
   :default-pathname "quark: code;"
   :default-package quark
   :default-module-type :fortran)
  (:serial "defs" "macros" (:parallel "things" "rooms") "parser"))
```

### **:journal-directory**

Specifies the location of the *journal* directory, which contains: the *system-directory* file and all of the *journal subdirectories*. See the section "Directories Associated with a System", page 191.

By default, the journal directory of a system is called the subdirectory "patch" under the default pathname. For example if the default directory is

```
sys: quux;
```

then the journal directory defaults to

```
sys: quux; patch;
```

Note: In order to convert pre-Genera 7.0 journal files into Genera 7.0 form: See the function `si:convert-journals` in *Converting to Genera 7.0*.

**:patchable**      `:patchable` specifies whether you want the system to be

patchable or not. It takes one argument, either `t` or `nil`. The default is `t`, meaning that the system is patchable. (See the section "Patch Facility", page 197.)

**:parameters** Specifies an "argument list" for the system. When you perform some operation on a system (compile or load it, for example), you can include extra keyword arguments that will be passed on to the methods that implement operations on the modules in the system. The value of **:parameters** is a list that reads like a keyword argument list.

Example: The **:parameters** option creates the keyword **:force-package** that can be passed on to system `foo` when it is compiled.

```
(defsystem foo
  (...
   :parameters (force-package))
  ...)
```

```
(compile-system 'foo :force-package 'foo-package)
```

In this example, the user-defined parameter **:force-package** keyword is not used by **compile-system** and is passed to the lower-level callee. In this example it could be the underlying compiler appropriate to the system being defined, like the Pascal compiler.

**:initializations** Creates a list of initializations to be run immediately after the last file in the system has been loaded. The format is **:initializations** *argument*. If *argument* is a symbol, it is interpreted as an initialization list. If it is an arbitrary form, it is evaluated.

This example specifies an initialization list:

```
:initializations *foo-init-list*
```

One of the files in your system, preferably the first one, should create the initialization list: (`defvar` *symbol* `nil`). For example:

```
(defvar *foo-init-list* nil)
```

You can add initializations to the list in your code. For example:

```
(add-initialization "init storage"
  '(setq *storage* nil) () '*foo-init-list*)
```

See the section "Introduction to Initializations" in *Internals, Processes, and Storage Management*.

- :initial-status** **:initial-status** *status* sets the initial status of the system when a new major version is created. The system's system-directory file records the status. The valid status keywords are **:experimental** (the default), **:broken**, **:obsolete**, and **:released**.
- :experimental** The system has been built but has not yet been fully debugged and released to users. The software is not stable.
- :released** The system is deemed stable and is released for general use.
- :obsolete** The system is no longer supported.
- :broken** The system does not work properly.
- :bug-reports** Specifies a list of two strings – (*list-name mouse-line-doc-string*). The first is the name of the bug mailing list to which bug reports are routed. Zmail uses this name in its Bug Mail menu. The second is a documentation string describing the purpose of the bug mail; the string appears in the mouse documentation line.
- Example: The following specification sends mail to Bug-Zmail.
- ```
:bug-reports ("Bug-Zmail" "Report problems with Zmail.")
```
- :advertised-in** Specifies a list of zero or more keywords indicating the contexts in which the system name and version number should be displayed. Valid keywords are:
- | <i>Keyword</i>     | <i>Meaning</i>                                                              |
|--------------------|-----------------------------------------------------------------------------|
| <b>:herald</b>     | The system name and version number are displayed in the herald.             |
| <b>:finger</b>     | The system name and version number are displayed in the Show Users listing. |
| <b>:disk-label</b> | The system name and version number are displayed in world load comments.    |

The default is **:herald**. Note that for a system not to appear in the herald, you must specify **:advertised-in ()**.

#### **:maintaining-sites**

**:maintaining-sites** (*site-list*) specifies the list of sites that maintain the system. **:maintaining-sites** declares the sites that can patch a system. It helps you to monitor versions in order to ensure that no changes are made an "unauthorized" sites. When you attempt to patch a system that is not maintained at your site, you receive a warning.

For example:

```
(defsystem experimental-file-system
 (...
  :maintaining-sites (:sgd :scrc))...)
```

The default for **:maintaining-sites** when it is undeclared is **nil**. This has the effect of allowing any site to patch the system without a warning.

**:source-category** The **:source-category** option is used for writing software distribution tapes. Its valid values are **:basic** (the default), **:optional**, and **:restricted**. These categories relate to distribution dumper categories. The distribution dumper writes out the sources for a system based on whether the system fits into the specified source-category. **:basic** is less restricted than **:optional**, which is less restricted than **:restricted**.

This option can also be specified as an alist, for example:

```
(:basic
 (:restricted "secrets" "more-secrets")
 (:optional "not-quite-as-secret"))
```

This says that all files are in the **:basic** category, except "secrets," "more-secrets," and "not-quite-as-secret."

#### **:distribute-sources**

The **:distribute-sources** option is used by the distribution dumper to decide whether or not to write sources to the distribution tape. It takes the values **t** or **nil**, and its default value is **t**.

#### **:distribute-binaries**

The **:distribute-binaries** option is used by the distribution dumper to decide whether or not to write binaries to the

distribution tape. It takes the values **t** or **nil**, and its default value is **nil**.

## 20.0.2 defsystem Modules

The body of a **defsystem** declaration names the files that compose the system and consists of one or more *module* specifications. A module is one or more files or modules that should be treated as a unit. *Operations* – compiling, loading, editing, hardcopying, and the like – are applied to the module as a whole.

Modules can be explicitly named or unnamed (*anonymous*). For example, in the long-form syntax,

```
(:module foo ("bar" "baz"))
```

is a named module called **foo** and contains two files – **bar** and **baz**. All operations are applied to the aggregate **foo**. The **:module** form names the aggregate (which the short-form **:parallel** would not do) and allows keyword modifiers to be associated with the module.

On the other hand, the following clause treats the files **bar** and **baz** as two separate but unnamed modules:

```
(:serial "foo" "bar")
```

A restriction on the construction of modules is that any one file in a module cannot *depend* on the operations performed on another file in that same module. If the compilation of file "bar" depends on file "baz" having been loaded, then these files cannot be placed in the same module.

A common organizing principle for grouping files into modules is to collect together those files that perform a similar function, with the restriction that the files within the module must not depend on one another. For example, all low-level definitions (variables and macros) might be placed in the same module.

Module specifications can be expressed using a long-form syntax, a simpler short-form syntax, or a hybrid of both formulations.

- Use the short form exclusively when your system uses only default types and packages and has straightforward dependency relationships.
- Use the long form as needed when your system contains component systems (i.e., when a module represents another system), non-default-type modules, explicit package specifications other than the system default package, or complicated dependency relationships.



### 20.0.2.1 Module Dependencies

Dependencies describe relationships among operations on modules. That is, they describe which modules depend on one another and for which operations they depend on one another. For example, modules often depend on the previous loading of other modules. The main program module in a system presumably depends on the previous loading of the low-level module definitions. Thus, the relationship of one **defsystem** module to another can be described as a hierarchy of dependencies. Within a module, however, no file can depend on any other file, but all files share the same dependencies vis-a-vis other modules.

Dependencies, which are described in the **defsystem** form, impose an order in which operations are performed on a module. The long-form module specification is needed to specify complicated dependencies among modules and operations. Note that a dependency does not guarantee that the operation will be performed, only that if the operation is requested (by the user), it will be performed in a certain order relative to other operations.

Formally defined, a module dependency states that under certain conditions, all specified operations must be performed on the indicated modules before the operation on the current module can take place.

#### Dependency Example 1

The following module specifications (assume they are Lisp modules) declare that:

- In order to load **main**, **defs** must be loaded first.

```
(defsystem foo
  ...
  (:module defs ("defs1"))
  (:module main ("main"))
  (:in-order-to :load (:load defs)))
```

The dependency in the example applies only when **foo** is loaded, and so is called a *load-time* dependency.

*Compile-time* dependencies, which apply only when a compile operation is performed, are slightly more complicated.

#### Dependency Example 2

Assuming that the **bar** system consists of Lisp-type modules, consider the **:in-order-to** clause below. This says that **macros** depends on the compilation and loading of **defs** whenever the **bar** system is compiled. At first glance, the compilation requirement is surprising because **(:load defs)**

does not mention anything about compilation. However, the system facility considers source files that can be compiled (such as Lisp or Pascal files) to have an *implicit* compile-time dependency on themselves: in order to load the files you must compile them first (if they are not already compiled).

Note: In order to prevent a Lisp file from being compiled at all, the two predefined module types `:lisp-read-only` and `:lisp-load-only` do not permit Lisp compilation.

```
(defsystem bar
  ...
  (:module defs ("defs"))
  (:module macros ("macros")
    (:in-order-to :compile (:load defs)))
  ...)
```

Dependencies can be expressed in different ways. Examples 1 and 2 declare the presence of a dependency relationship explicitly. A module can also describe a dependency implicitly using a short-form syntax.

### Dependency Example 3

The `:serial` clause implies that `main` depends on `defs` and that `defs` does not depend on any other module. It also implies that operations on "defs" and "main" be performed separately and in order, even though it does not explicitly state these operations. So, if a compile operation were performed on system `foo`, first `defs` would be compiled and loaded, then `main` would be compiled and loaded.

```
(defsystem foo
  ...
  (:serial "defs" "main"))
```

In Examples 1 and 2, `defs` contains only one file, "defs", but if `defs` consisted of two files, "defs1" and "defs2", then the examples would have to be rewritten. This is relatively straightforward for Example 1; the single module specification would be edited as follows:

```
(:module defs ("defs1" "defs2"))
```

All operations would be applied to the aggregate `defs`.

Changing Example 3 requires altering the dependency to say that "defs1" and "defs2" do not depend on one another. However, `main` still depends on the prior compilation/loading of "defs1" and "defs2" but in no particular order. This dependency would be written like so:

```
(:serial (:parallel "defs1" "defs2") "main")
```

The embedded **:parallel** clause declares that the files that follow have no dependency relationship; they are operated on as a unit. The **:serial** clause still states that any operations are applied first to the **:parallel** clause, then to **main**.

Once you correctly determine (1) which files should compose a module and (2) which and how modules depend on one another, you will never have to figure out these relationships again. By constructing a plan based on the modules and their dependencies, you have finished your part of the job. Commands that operate on systems, like Load System, will work correctly.

### 20.0.2.2 Short-form Module Specifications

Short-form specifications provide an abbreviated syntax for defining groups of unnamed (anonymous) modules that have a straightforward dependency relationship. All the system's files must be of the default type (defined by the **:default-module-type** option) if they are named explicitly in the short-form specification.

A short-form specification consists of a keyword, followed by one or more elements: (*keyword element1 element2 ...*)

An *element* can be another short-form or a *primary*. A primary is either a symbol, which is interpreted to be the name of a named module, or a string, which is a file spec.

The *keyword* describes the dependency relationship among the modules and can be any of the following: **:serial**, **:parallel**, **:definitions**, or **:module-group**.

Short forms can be embedded in short forms.

The meanings of the keywords are explained here.

- **:serial** means that each of the specified elements depends in some way on the preceding one. The order of specification is therefore essential.

**Example:** If the compile operation is performed on the system, each Lisp module in the clause shown below is compiled and then loaded in turn before the next one is compiled and loaded. The compilation and loading of **glub** depends on the previous compilation and loading of **bar**. In order to compile and load **bar**, the computer must have already compiled and loaded **foo**.

```
(:serial "foo" "bar" "glub")
```

- **:parallel** means that the specified elements do *not* depend on one another in any way; they are operated on as a group. The order of specification is therefore not important.

Example: If the compile operation is performed, all the Lisp modules in the following clause are compiled, then all are loaded.

```
(:parallel "foo" "bar" "glub")
```

- The syntax of the **:definitions** clause is `(:definitions primary element)`. **:definitions** means that the *element* has a *serial* dependency on the *primary* and, in addition, it has a compile-dependency. This means that if the *primary* is compiled, the *element* must be compiled. The **:definitions** clause is useful when the *primary* contains macros that are used in the definition of the *element*.
- **:module-group** is an additional short-form syntax keyword. It provides a way to name the aggregate result of a short-form specification, so that other specifications can refer to this result. The format is: `(:module-group name short-form options)`.

The structure is analogous to, and the options are the same as for, the long-form specification. See the section "Long-form Module Specifications", page 158.

### Short Form Syntax Examples

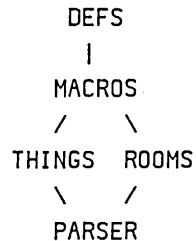
The following short-form syntax **defsystem** illustrates serial dependency with an embedded parallel dependency.

```
(defsystem adventure
  (:default-pathname "quark: code;"
   :default-package quark
   :default-module-type :fortran)
  (:serial "defs" "macros" (:parallel "things" "rooms") "parser"))
```

The **adventure** system consists of a sequence of modules of the type **:fortran**, compiled in the **quark** package. In the event that the system is compiled, then operations occur as follows:

1. Compile **defs**, then load it
2. Compile **macros**, then load it
3. Compile **things** and **rooms**, then load both of them
4. Compile and load **parser**

The following diagram illustrates the above dependency relationship.



Both "things" and "rooms" depend on "defs" and "macros" to have been compiled and loaded, but "things" and "rooms" do not depend on each other with respect to compilation. "Parser" depends on "things" and "rooms" having been compiled and loaded but in no particular order.

The next example shows how the **:module-group** keyword is used. The **:module-group** names the result of the included short-form specification **bigstuff**, so that the **main** module can refer to it as a dependency: in order to compile **main**, first compile and load **bigstuff**.

```

(:module-group bigstuff
  (:definitions "macros" (:parallel "foo" "bar" "blech")))
(:serial bigstuff (:parallel "a" "b" "c"))
  
```

### 20.0.2.3 Long-form Module Specifications

Use the long-form module specification when your system contains component systems, non-default-type modules, explicit package specifications other than the system default package, or complicated system dependencies.

The general format of a long-form specification is:

```

(:module name inputs
  (keyword-option-1)
  (keyword-option-2)
  ... )
  
```

The **:module** keyword defines the module called *name*. *name* must be a symbol or **nil**; **nil** means that the module is anonymous.

*inputs* can be **nil** or a list of one or more of the following:

- Strings representing a file name
- Symbols representing the name of another system defined by **defsystem**

When a module consists of more than one input, the inputs must be specified as a list.

The ordering of inputs *within* a module specification is not significant. Dependencies are determined by explicit keyword directives in **:module** clauses or, failing that, by the order of the modules in the system declaration.

### **:module Keyword Options**

**:package** and **:type** override the system defaults for package and types, respectively.

**:package** The **:package** option takes one argument, a string, and causes operations on a module to be performed in the specified package. It overrides both the system default (specified by the **:default-package** option to **defsystem**) and any package named in the attribute lists of the system's files. It does not override the **:package-override** option to **defsystem**.

Example: The **macros** module is compiled and loaded into the **special** package. All other modules are compiled and loaded into the system default, **mail**.

```
(defsystem mailer
  (...
   :default-package mail
   ...))
(:module defs "defs")
(:module macros "macros" (:package special)
  (:in-order-to :compile (:load defs)))
...)
```

**:type** The **:type** option in a module specification overrides the default module type for the system. The type specifies the nature of the inputs to that module, for example, whether it's composed of Pascal files, Lisp files, or ordinary text files, and determines the details of what is done for each generic operation (for example, load, edit, hardcopy) performed on that module. Each type has certain valid operations. You can use any of the predefined types, including **:lisp**, **:text**, **:font**, **:lisp-example**, **:system**, and so on. See the section "Table of Module Types and Operations", page 169. You can also define your own module types. See the section "User-defined Module Types", page 171.

Example 1: The inputs to **adventure1** are all Fortran files;

however, if the parser had been written in Lisp, then the **defsystem** form should be rewritten as shown in **adventure2**. **parser** is explicitly declared to be a module of type **:lisp**.

```
;;; Example 1
(defsystem adventure1
  (:short-name "advent1"
   :default-pathname "quark: code;"
   :default-package quark
   :default-module-type :fortran)
  (:serial "defs" "macros" (:parallel "things" "rooms") "parser"))

(defsystem adventure2
  (:short-name "advent2"
   :default-package quark
   :default-pathname "quark: code;"
   :default-module-type :fortran)
  (:module parser ("parser") (:type :lisp))
  (:serial "defs" "macros" (:parallel "things" "rooms") parser))
```

The **:system** type specifies the names of *component systems*, which are other systems (defined by a **defsystem** or **defsubsystem** form) that are to be included in this system. System operations are performed recursively. In the usual case, performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems.

Example 2: The moderately complicated definition of **common-lisp-internals** falls rather gracefully and readably into the serial-parallel abbreviated form. Then **common-lisp-internals** is easily made a component system of **common-lisp** by designating it as module **cl** of type **:system**. Note how neatly a compile and load dependency on **cl** is specified in the **:serial** clause.

```

;;; Example 2
(defsubsystem common-lisp-internals
  (:default-pathname "sys:clcp;"
   :default-package cli)
  (:serial "functions" "sequence-macros" "numerics"
   (:parallel "listfns" "seqfns" "hashfns")
   "type-infra" "type-supra" "type-supra2" "Type-supra3"
   "More-functions" "Stringfns" "Charfns" "Arrayfns" "Error"
   (:parallel "Iofns" "Read-print")))

(defsubsystem common-lisp
  (:default-pathname "sys:clcp;")
  (:module cl common-lisp-internals (:type :system))
  (:serial cl "Permanent-links"))

```

**:in-order-to** and **:uses-definitions-from** are the two main options for controlling the dependency relationships among modules.

**:in-order-to**

**:in-order-to** is the basic keyword that expresses dependency relationships among modules. The general format is

**(:in-order-to (:operation-1 :operation-2 ...) (:operation module))** Note that the first argument to **:in-order-to** can be either a symbol or a list.

Example: The following code fragment illustrates a compile-time and a load-time dependency.

```

(:module main ("main")
  (:in-order-to :compile (:load defs))
  (:in-order-to :load (:load utils)))

```

It directs that:

- If the compile operation is performed on the present module, **main**, then the **defs** module must be loaded first.
- If **main** is loaded, then the module **utils** must be loaded first.

**:uses-definitions-from**

**:uses-definitions-from** is similar to the **:in-order-to** option. The general format is **(:uses-definitions-from module)**.



Writing **(:uses-definitions-from foo)** implies the dependency relation:

```
(:in-order-to (:compile :load) (:load foo))
```

To state it another way, **:uses-definitions-from** means that the module has a **serial** dependency on the depended-upon *module*. In addition, it requires that if the depended-upon module needs to be recompiled, then all of its dependents will be recompiled as well. Note that dependencies are transitive.

Example: Consider the following fragment, assuming that the **macros** module has been defined in the **defsystem** form.

```
(defsystem jonathan
  (:default-pathname "sys:jonathan;"
   :default-package c1)
  (:module macros ("bim" "bam" "boom"))
  (:module A ("a" "b" "c")
   (:uses-definitions-from macros))
```

The **:uses-definition-from** clause affects module **A** in the following ways:

- If **macros** is being compiled, then compile **A** whether or not it is otherwise necessary.
- If **A** is being compiled, then compile **macros**, if it needs to be compiled, first.
- If **A** is being loaded, then load **macros**, if it needs to be loaded, first.

**:root-module** and **:compile-satisfies-load** also control the order in which operations are performed but are far less commonly used than **:in-order-to** and **:uses-definitions-from**.

**:root-module** The **:root-module** option is useful for controlling the loading and compilation of macro definitions. It has the effect of altering the normal rules of dependency. Its valid values are:

| <i>Value</i> | <i>Meaning</i>                                                                                                 |
|--------------|----------------------------------------------------------------------------------------------------------------|
| t            | Designates the indicated module as a <i>root module</i> – a module that is always processed. t is the default. |

**nil** Indicates that the module is not a root module.

This attribute affects system building as follows: when a command or function that operates on a system (like Compile System) constructs a step-by-step plan to operate on a system (compiling, loading, as necessary) it will not include a step for a non-root-module *unless it is explicitly depended upon by another module*. That is, compilation (or loading, and so on) of this module occurs only if a dependency exists.

Example: In the following example, the **macros** module specifies that it should not be considered a root module.

```
(defsystem rm-example
  (:default-pathname "example: code;")
  (:module defs ("defs"))
  (:module macros ("macros")
    (:in-order-to :compile (:load defs))
    (:root-module nil))
  (:module utils ("utils")
    (:uses-definitions-from macros)
    (:in-order-to :compile (:load macros))
    (:in-order-to :load (:load defs)))
  (:module main ("main")
    (:uses-definitions-from macros)
    (:in-order-to :compile (:load macros))
    (:in-order-to :load (:load utils))))
```

Assuming that the user has requested a system load, examine the load-time dependencies and note that, for purposes of loading, **macros** is *not* depended upon by any other module:

- **defs** does not depend on any other module
- **macros** depends on **defs** being loaded
- **utils** depends on **defs** being loaded
- **main** depends on **utils** being loaded

Thus, **macros** is ignored during the preparation of the system construction plan for loading **rm-example**:

1. Load **defs**

2. Load **utils**

3. Load **main**

If **:root-module** had not been specified or had been given a value of **t**, **macros** would have been loaded, according to the normal dependency rules. Since macro definitions need not be installed when a system is being loaded to be used, **(:root-module nil)** gives exactly the desired result.

When the same system is compiled, however, a load of **macros** is included in the system construction plan because **macros** is depended upon at compile-time by two modules.

- **defs** does not depend on any other module
- **macros** depends on **defs** being compiled, if necessary, and loaded
- **utils** depends on **macros** being compiled, if necessary, and loaded
- **main** depends on **macros** being compiled, if necessary, and loaded

Since macro definitions need only be loaded at compile-time, **(:root-module nil)** again gives exactly the desired result.

Note: for specifications in the old style (pre-Genera 7.0) that included the **:skip** directive, use the new directive **(:root-module nil)** inside the module declaration.

### **:compile-satisfies-load**

The **:compile-satisfies-load** option, like **:root-module**, is useful for controlling the compilation and loading of macro definitions and alters the normal rules of dependency.

It has two valid values: **t** and **nil**. When set to **t**, the option declares that when a module is compiled in the current compiler environment, it should not be loaded – even if a load dependency exists, because the loading the module could destroy the current environment. The load dependency is satisfied by compiling the module.

When set to **nil**, **:compile-satisfies-load** specifies that when a module is compiled in the current compiler environment, load it if necessary. **nil** is the default.

This feature is useful because the compiler will notice entities like **defmacro**, **defsubst**, **zl:defstruct**, and **defflavor** and use them for the compilation of subsequent files without having to load them. However, if the bodies of macros (not the code produced by their expansion) call subroutines (**defuns**) in the file, then the file must be loaded in order to define those subroutines.

**Example of :compile-satisfies-load:** Assume that the user has requested a compile of the **cs1-example** system.

```
(defsystem cs1-example
  (:default-pathname "example: code;")
  (:module defs ("defs"))
  (:module macros ("macros")
    (:in-order-to :compile (:load defs))
    (:root-module nil)
    (:compile-satisfies-load t))
  (:module utils ("utils")
    (:uses-definitions-from macros)
    (:in-order-to :compile (:load macros))
    (:in-order-to :load (:load defs)))
  (:module main ("main")
    (:uses-definitions-from macros)
    (:in-order-to :compile (:load macros))
    (:in-order-to :load (:load utils))))
```

The compile-time dependencies expressed above indicate that:

- **defs** does not depend on any other module
- **macros** depends on **defs** being compiled, if necessary, and loaded
- **utils** depends on **macros** being compiled, if necessary, and loaded
- **main** depends on **macros** being compiled, if necessary, and loaded

If the **:compile-satisfies-load** attribute were absent or set to **nil** the system construction plan would look like this:

1. Compile **defs**
2. Load **defs**
3. Compile **macros**

4. Load **macros**
5. Compile **utils**
6. Compile **main**
7. Load **utils**
8. Load **main**

Note that because the **:compile-satisfies-load** attribute is present, the plan is amended to delete step 4.

The **:source-category**, **:distribute-sources**, and **:distribute-binaries** options supply values that override within the module the corresponding default values for the system.

**:source-category** The **:source-category** option is used for writing software distribution tapes. Its valid values are **:basic** (the default), **:optional**, and **:restricted**. These categories relate to distribution dumper categories. See the section "Distribution Dumper". The distribution dumper writes out the sources for a system based on whether the system fits into the specified source-category. **:basic** is less restricted than **:optional**, which is less restricted than **:restricted**.

This module option can also be specified as an alist. See the **:source-category** option to **defsystem**.

#### **:distribute-sources**

The **:distribute-sources** option is used by the distribution dumper to decide whether or not to write sources to the distribution tape. It takes the value **t** or **nil**, and its default value is **t**.

#### **:distribute-binaries**

The **:distribute-binaries** option is used by the distribution dumper to decide whether or not to write binaries to the distribution tape. It takes the values **t** or **nil**, with a default value of **nil**.

#### **sct:undefsystem** *system-name*

*Function*

Removes all record of the system called *system-name* from **sct:\*all-systems\*** and removes all the source file name properties from the system. The effect is to make it look like a **defsystem** wasn't even done. Note: This does not undefine functions, flavors, etc., created by loading the system.

**defsubsystem** *system-name options &body body* *Special Form*

Defines a system that has no autonomous existence and is not patchable. It can only be compiled and loaded by compiling or loading its parent system. It can, however, be treated independently for some operations, like edit or hardcopy.

In a **defsystem** form, a subsystem is specified as a **:module** and is flagged with the keyword pair (**:type system**) (see the example). Subsystems are provided as a convenience for specifying groups of modules that are all in one package or directory. Subsystems have no associated component directory. Their files are journaled in the parent system's component directory.

Subsystems retain identity as systems on which you can select as a tag table in Zmacs.

In the following example of **defsubsystem**, we have not listed all the file names for each system and subsystem. The places where these names normally go are marked by ellipses.

```
(defsystem fortran
  (:default-pathname "sys: fortran;"
   :journal-directory "sys: fortran;"
   :patchable t)
  (:module macros ("macros") (:root-module nil))
  (:module language-tools (language-tools) (:type :system))
  (:module front-end (fortran-front-end) (:type :system))
  (:module back-end (fortran-back-end) (:type :system))
  (:serial macros language-tools front-end back-end))

;;; Component system definition
(defsystem language-tools
  (:default-pathname "sys: language-tools;"
   :patchable t)
  (:serial ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-front-end
  (:default-pathname "sys: fortran;")
  (:serial "tokenizer" "grammar" ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-back-end
  (:default-pathname "sys: fortran;")
  (:serial "code-generator" "optimizer" ... ))
```

In the example, **language-tools** is a patchable component system, and **fortran-front-end** and **fortran-back-end** are both subsystems.

## 20.1 defsystem Operations

With the Genera 7.0 **defsystem**, specifications of modules are intermingled with operations on modules. This stands in contrast to the syntax of **defsystem** in earlier releases in which module clauses and "transformation" clauses were separate.

This section gives a brief overview of the kinds of operations that can be applied to systems. For more details on these operations, see the referenced sections.

Seven types of predefined operations are available:

|              |                                                                                                                                                                                                                                                            |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Load         | Load the system into the current environment. Invoked by the Command Processor command Load System and the function <b>load-system</b> . See the section "Loading and Compiling Systems", page 175.                                                        |
| Compile      | Compile the system, create journal files, and optionally load it into the current environment. Invoked by the Command Processor command Compile System and the function <b>compile-system</b> . See the section "Loading and Compiling Systems", page 175. |
| Edit         | Read all the files of the system into editor buffers. Invoked by the Command Processor command and the function <b>sct:edit-system</b> . See the section "Other Operations on Systems", page 187.                                                          |
| Hardcopy     | Hardcopy all the files in the system. Invoked by the function <b>sct:hardcopy-system</b> . See the section "Other Operations on Systems", page 187.                                                                                                        |
| Reap-protect | Reap-protect the system. This marks all source and product files as protected from deletion. Invoked by the Command Processor command and the function <b>sct:reap-protect-system</b> . See the section "Other Operations on Systems", page 187.           |
| Distribute   | Write the system on tape. Invoked by the Command Processor command Distribute Systems (note the plural form, since one or more systems can be written to tape at a time). See the section "Other Operations on Systems", page 187.                         |
| Release      | Puts the <b>:released</b> keyword in the system's patch directory, and                                                                                                                                                                                     |

inserts a **:released** designation in the system directory file. For most operations on a system, the **:released** designator is used as the default version. Failing this, the **:latest** version is used. Invoked by the function **sct:release-system**; no corresponding Command Processor command. See the section "Other Operations on Systems", page 187.

### 20.1.1 Table of Module Types and Operations

This is a table of system module types and their behavior under standard operations. Note: The operation **sct:reap-protect** applies to all types of systems and so is not listed here. See the legend below the table to find the meaning of the various abbreviations used.



| Module Type          | Default file type | Compile                     | Load      | Hard-copy | Edit | Distribute (source/product) |
|----------------------|-------------------|-----------------------------|-----------|-----------|------|-----------------------------|
| Lisp                 | :lisp             | L-comp                      | BL        | T         | T    | T/T                         |
| Prolog               | :prolog           | P-comp                      | BL        | T         | T    | T/T                         |
| Ada                  | :ada              | A-comp                      | BL        | T         | T    | T/T                         |
| Fortran              | :fortran          | F-comp                      | BL        | T         | T    | T/T                         |
| Pascal               | :pascal           | Pa-comp                     | BL        | T         | T    | T/T                         |
| Text                 | :text             | --                          | --        | T         | T    | T/--                        |
| Font                 | :bfd              | --                          | FL        | N         | N    | T/--                        |
| System               | --                | *** Operate recursively *** |           |           |      |                             |
| Lisp-example         | :lisp             | --                          | --        | T         | T    | T/T                         |
| Readtable            | :lisp             | R-comp                      | BL        | T         | T    | T/T                         |
| Lisp-read-only       | :lisp             | --                          | Read-file | T         | T    | T/--                        |
| Lisp-load-only       | :lisp             | --                          | BL        | T         | T    | --/T                        |
| Logical-translations | :lisp             | --                          | Read-file | T         | T    | T/--                        |
| Binary-data          | :bin              | --                          | --        | N         | N    | --/T                        |
| Text-data            | :text             | --                          | --        | T         | T    | T/--                        |

Legend: "i-comp" means the appropriate compiler is used, e.g., "L-comp" means the Lisp compiler is invoked. "--" means this operation is meaningless on this file type. "BL" means the binary loader is invoked. "FL" refers to the font loader.

Besides the standard, predefined operations, you can define your own operations on modules. See the section "User-defined Operations on Systems", page 172.

### 20.1.2 System Plan

The order in which operations are performed on the modules in a system is called the *system plan*. By default, operations occur in the order that they are defined or they are shuffled the minimum amount necessary to realize the specified constraints. These constraints are in the form of dependencies (i.e., module X must be loaded before module Y is loaded).

In order to see in advance the system plan for a given system with a given operation, type the Command Processor command: `Show System Plan name-of-system (operation)`. Two factors determine the system plan:

1. The order in which the modules are defined
2. The ordering constraints that derive from the dependencies that are specific to that operation

#### Show System Plan Command

Show System Plan *system operation*

Show the system plan (i.e., the order of operations) for the specified *system* under the specified *operation*. *operation* can be All, Compile, Load, Edit, Hardcopy, Reap-protect, or Distribute.

## 20.2 User-defined Module Types

You can define your own module types using the function `sct:define-module-type`.

`sct:define-module-type type source-default product-default &body Function`  
*base-flavors*

Defines a new module type called *type* with a *source-default* module type and a *product-default* module type.

The *base-flavors* are the previously defined module type upon which this type is built. The new *type* inherits the properties of the *base-flavors* and interprets operations like the *base-flavors* do, except in the case that special methods are defined for the *type* that override the *base-flavors* operations.

Once you have defined a module type, you define methods with `defmethod` that implement the special behavior of the new module type for the standard operations: compile, load, and so on.

The purpose of this example is to define a module type called **lisp-read-only** whose sources are Lisp code but which is meant to be read and not compiled. According to the definition of **lisp-read-only** in the example, a module of this type will respond according to the definition of its base flavor **lisp-module** for all operations except loading and compiling.

```
(define-module-type :lisp-read-only :lisp nil
  lisp-module)

(defmethod (:compile lisp-read-only-module) (system-op &rest keys)
  (ignore system-op keys)
  nil)

(defmethod (:load lisp-read-only-module) (system-op &rest keys)
  (lexpr-send self :read system-op keys))
```

### 20.3 User-defined Operations on Systems

It is usually more useful to define your own type of system module than it is to define your own operation. However, SCT provides a facility for defining your own operations, should you need it. The macro **sct:define-system-operation** is the primary tool for this purpose.

```
sct:define-system-operation operation driving-function documentation Macro
  &key (arglist '(system-name &key
    query :confirm silent batch (version :latest)
    (include-components t) &rest keys
    &allow-other-keys)) (class :normal)
    (subsystems-ok t) body-wrapper (encache :both)
```

Defines a manipulation called *operation* to be applied to a system, creating a function called *operation-system*. The *driving-function* is a closure – the operation itself at the level of what is done to a single file. Higher-level mechanisms take care of applying this operation to each file in a system. The *documentation* is another closure – an operation that prints what will be done to the file. The *arglist* specifies the arguments that are accepted by the operation. The operation can also process the keyword arguments **:query**, **:batch**, **:version**, and **:include-components**. For the meaning of these keywords: See the function **load-system**, page 176.

The *encache* argument is used by SCT to optimize calls to **fs:multiple-file-plists**. Typically, you should use **:both** if the operation needs to look at any file properties (e.g., compile) or **nil** if the operation does not need to look at any properties (e.g., edit or hardcopy). *class* should

be **:normal** for operations that construct a plan according to dependencies (e.g., compile, load, edit) or should be **:simple** for operations that work on everything in the system (e.g., reap-protect).

The definition of the standard hardcopy operation is shown next as an example of the use of the **sct:define-system-operation** macro.

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: SCT; Base: 10 -*-

(define-system-operation :hardcopy
  ;      input output module      keywords
  (lambda (source ignore ignore &rest ignore)
    (declare (special hardcopy:*default-text-printer*))
    (hardcopy:hardcopy-file source hardcopy:*default-text-printer*))
  ;      input output module      keywords
  (lambda (source ignore ignore &rest ignore)
    (format standard-output "~&Hardcop~[y~;ying~;ied~] file ~A"
      *system-pass* source))
  :arglist
  (system-name &key (query :confirm) silent batch
    (include-components t) (version :newest)
    &rest keys &allow-other-keys)
  :encache nil
  :class :normal)
```



## 21. Loading and Compiling Systems

The **load-system** and **compile-system** forms, with their Command Processor equivalents Load System and Compile System, are the means of loading and compiling systems. These functions replace the function **make-system** that was used for these purposes in Symbolics Release 6.1 and earlier releases.

### Load System Command

Load System *system keywords*

Loads a system into the current world.

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>system</i>     | Name of the system to load. The default is the last system loaded.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <i>keywords</i>   | :Condition :Load Patches :Query :Redefinitions Ok :Silent :Simulate :Version                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| :Condition        | {always, never, newly-compiled} Under what conditions to load each file in the system. Always means load each file. Newly-compiled means load a file only if it has been compiled since the last load. The default is newly-compiled.                                                                                                                                                                                                                                                                                                                        |
| :Load Patches     | {yes, no} Whether to load patches after loading the system. The default is yes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| :Query            | {Everything, Confirm-only, No} Whether to query before loading. Everything means query before loading each file. Confirm-only means create a list of all the files to be loaded and then ask for confirmation before proceeding. No means just go ahead and load the system without asking any questions. The default is No. The mentioned default is Everything.                                                                                                                                                                                            |
| :Redefinitions Ok | {yes, no} Controls what happens if the system asks for confirmation of any redefinition warnings during the loading process. Yes means assume that all requests for confirmation are answered yes and proceed. No means pause at each redefinition and await confirmation. The default is No. The mentioned default is Yes. This allows you to start loading a system that you know will take a long time to load and leave it to finish by itself without interruption for questions such as "Warning: <i>function-name</i> being redefined, ok? (Y or N)". |

|                  |                                                                                                                                                                                              |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>:Silent</b>   | {yes, no} Whether to turn off output to the console while the load is going on. The default is no. Adding this keyword to your Load System command string is the same as <b>:silent</b> yes. |
| <b>:Simulate</b> | {yes, no} Print a simulation of what compiling and loading would do. The default is no. Adding this keyword to your Load System command string is the same as <b>:simulate</b> yes.          |
| <b>:Version</b>  | {released, latest, newest, <i>version-number</i> , <i>version-name</i> } Which version number to load. The default is released.                                                              |

Note: This command only loads a system. If you want to compile and load a system: See the section "Compile System Command", page 177.

**load-system** *system-name* &key (*query* **:confirm**) (*silent* **nil**) (*batch* **nil**) (*no-warn* **nil**) (*reload* **nil**) (*no-load* **nil**) (*never-load* **nil**) (*dont-set-version* **nil**) (*include-components* **t**) (*load-patches* **t**) (*version* **:released**) &allow-other-keys *Function*

Loads the system named by *system-name* into the current environment, according to the specified keyword options.

### 21.0.1 load-system Keywords

These are the predefined keyword options to **load-system**. Note that the allowable keywords can include those declared in the **:parameters** part of the **defsystem**.

|                 |                                                                                                                                                                                                                                                                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>:query</b>   | Takes <b>t</b> , <b>nil</b> , <b>:confirm</b> , or <b>:no-confirm</b> . If <b>t</b> , ask for approval of each and every operation. If <b>nil</b> or <b>:no-confirm</b> , don't ask about anything. If <b>:confirm</b> , list all the operations and then ask for confirmation. Default-value: <b>:confirm</b> .                                     |
| <b>:silent</b>  | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , perform all operations without printing anything. If <b>:query</b> is non- <b>nil</b> , <b>:silent</b> <b>t</b> is overridden. Default value: <b>nil</b> .                                                                                                                                              |
| <b>:no-warn</b> | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , don't bother to print a redefinition warning when a function is redefined. Default value: <b>nil</b> .                                                                                                                                                                                                  |
| <b>:batch</b>   | Takes <b>t</b> , <b>nil</b> , or <i>pathname</i> . Simulate <b>:query</b> <b>:confirm</b> <b>:silent</b> <b>t</b> <b>:no-warn</b> <b>t</b> and collect the compiler warnings and write them to <i>system-name.cwarns</i> . If <i>pathname</i> , do the same as <b>t</b> but write compiler warnings to <i>pathname</i> . Default value: <b>nil</b> . |

- :reload** Takes **t** or **nil**. If **t**, reload all the product (.bin) files, even if the version in the environment is the most recent version. Default value: **nil**.
- :no-load** Takes **t** or **nil**. If **t**, do not load .bin files unless they are required by a specific dependency in the **defs** system. Default value: **nil**.
- :never-load** Takes **t** or **nil**. If **t**, never load any .bin files, no matter what dependencies say. Default value: **nil**.
- :include-components** Takes **t** or **nil**. If **t**, perform the requested system operation on component systems. Default value: **t**.
- :load-patches** Takes **t** or **nil**. After the system has been loaded, implicitly perform a **load-patches** operation. Default value: **t**.
- :version** Takes **:latest**, **:newest**, **:released**, a number, or another designator. **:latest** means the latest major version recorded in the journal directory. **:newest** means ignore the journal directory and find the newest version of the files.
- :dont-set-version** Takes **t** or **nil**. If **t**, do not worry about setting the version number of the system in the running world. This is an optimization used to speed up the loading of some systems such as the Logical Pathname Translations Files system. Default-value: **nil**.

## Compile System Command

Compile System *system* keywords

Compile the files that make up *system*.

- system-spec* name of the system to compile. The default is the last system loaded.
- keywords* **:Batch**, **:Condition**, **:Load**, **:New Major Version**, **:Query**, **:Redefinitions Ok**, **:Silent**, **:Simulate**, **:Update Directory**
- :Batch** {yes, no} Whether to save the compiler warnings in a file instead of printing them on the console. The default is no, to just print them on the console. Adding the keyword **:batch** to your Compile System command is the same as **:batch yes**.
- :Condition** {always, new-source} Under what conditions to compile each file



- in the system. Always means compile each file. New-source means compile a file only if it has been changed since the last compilation. The default is new-source.
- :Load** {Everything, Newly-Compiled, Only-For-Dependencies, Nothing} Whether to load the system you have just compiled into the world. The default is Newly-Compiled.
- :New Major Version** {yes, no, ask} Whether to give your newly compiled version of the system the next higher version number. The default is yes. Giving the choice no will ask you to confirm that you really want to "prevent incrementing system major version number".
- :Query** {Everything, Confirm-only, No} Whether to query before compiling. Everything means query before compiling each file. Confirm-only means create a list of all the files to be compiled and then ask for confirmation before proceeding. No means just go ahead and compile the system without asking any questions. The default is No. The mentioned default is Everything.
- :Redefinitions Ok** {yes, no} Controls what happens if the system asks for confirmation of any redefinition warnings during the compilation. Yes means assume that all requests for confirmation are answered yes and proceed. No means pause at each redefinition and await confirmation. The default is No. The mentioned default is Yes. This allows you to start a compilation that you know will take a long time and leave it to finish by itself without interruption for questions such as "Warning: *function-name* being redefined, ok? (Y or N)".
- :Silent** {yes, no} Whether to suppress output to the console. The default is no, to allow output. Adding the :silent keyword to your Compile System is the same as :silent yes.
- :Simulate** {yes no} Print a simulation of what compiling would do. The default is no. Adding this keyword to your Compile System command string is the same as :simulate yes.
- :Update Directory** {yes, no} Whether to update the directory of the system's components. The default is yes.
- compile-system** *system-name* &key (*query :confirm*) (*silent nil*) (*batch nil*) (*no-warn nil*) (*recompile nil*) (*no-compile nil*) (*reload nil*) (*no-load nil*) *Function*

(*never-load* **nil**) (*increment-version* **t**)  
 (*update-directory* **t**) (*initial-status* **nil**)  
 (*include-components* **t**) (*load-patches* **t**) (*version*  
**:newest**) &allow-other-keys

Compiles the system named by *system-name* with the specified keyword options.

### 21.0.2 compile-system Keywords

These are the predefined keyword options to **compile-system**. Note that the allowable keywords can include those declared in the **:parameters** part of the **defsystem**.

- :recompile** Takes **t** or **nil**. If **t**, recompile all the source files, even if the **.bin** file is newer than the source file. Default value: **nil**.
- :no-compile** Takes **t** or **nil**. If **t**, do not compile any source files, no matter what anyone else says. This is useful in conjunction with **:update-directory t** and **:increment-version nil**, since it buys the ability to fix up the journal files after you have hand-compiled some source files. Default value: **nil**.
- :increment-version**  
 Takes **t** or **nil**. If **t**, create a new major version number. Default value: **t**.
- :update-directory**  
 Takes **t**, **nil**, or *keyword*. If **t**, update the journal files. If *keyword*, update the journal files and add a designator of *keyword* for the newly created version. Furthermore, if *keyword* is **:released**, then declare the status of the system to be released. Default value: **t**.
- :initial-status** Takes *keyword*. Declare the initial status of the system to be *keyword*. Default value: **:experimental**.
- :query** Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each and every operation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.
- :silent** Takes **t** or **nil**. If **t**, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.

|                            |                                                                                                                                                                                                                                                                                                                   |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>:no-warn</b>            | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , don't bother to print a redefinition warning when a function is redefined. Default value: <b>nil</b> .                                                                                                                                                               |
| <b>:batch</b>              | Takes <b>t</b> , <b>nil</b> , or <i>pathname</i> . Simulate <b>:query :confirm :silent t :no-warn t</b> and collect the compiler warnings and write them to <i>system-name.cwarns</i> . If <i>pathname</i> , do the same as <i>t</i> but write compiler warnings to <i>pathname</i> . Default value: <b>nil</b> . |
| <b>:reload</b>             | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , reload all the product (.bin) files, even if the version in the environment is the most recent version. Default value: <b>nil</b> .                                                                                                                                  |
| <b>:no-load</b>            | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , do not load .bin files unless they are required by a specific dependency in the <b>defsystem</b> . Default value: <b>nil</b> .                                                                                                                                       |
| <b>:never-load</b>         | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , never load any .bin files, no matter what dependencies specify. Default value: <b>nil</b> .                                                                                                                                                                          |
| <b>:include-components</b> | Takes <b>t</b> or <b>nil</b> . If <b>t</b> , perform the requested system operation on component systems. Default value: <b>t</b> .                                                                                                                                                                               |
| <b>:load-patches</b>       | Takes <b>t</b> or <b>nil</b> . After the system has been loaded, implicitly perform a <b>load-patches</b> operation. Default value: <b>t</b> .                                                                                                                                                                    |

## 21.1 Loading System Definitions That Use Logical Pathnames

Once you have written a large program and defined it as a system, you want **load-system** (or the Command Processor commands Load System and Compile System) to compile and load the system and any patches. Assuming that your system definition uses logical pathnames, you must write these three files for **load-system** to be able to find and load your system:

- System file, named *sys:site;system-name.system* file. Note: This is useful for systems that are meant to be generally available to others at your site. Experimental or private systems need not be defined in a separate *system* file. You can compile the **defsystem** in an editor buffer or put a form that loads the system declaration in your initialization file.
- Translations file, named *sys:site;logical-host.translations* file. This is only necessary in the special case in which a logical host has been specifically defined for this system.
- System declaration file, commonly named

*logical-host:logical-directory;system-name.lisp* or  
*logical-host:logical-directory;sysdcl.lisp*

The `sys:site;` logical directory is the repository for all systems, those you define and those distributed by Symbolics. When a world load is transported to a new site, the translation file for each logical host that is defined in the current world is reloaded from the new site's `sys:site;` directory. In this way, all logical pathnames are mapped into the set of physical pathnames defined at the new site.

### 21.1.1 `sys:site;`System-name.System File

`load-system` looks in the `sys:site;` logical directory for the *system-name.system* file (the system file) when given a system name that is undefined in your environment. For example, if you type `(load-system 'graphic-lisp)` it looks for the file `sys:site;graphic-lisp.system`.

The system file contains the form:

```
(sct:set-system-source file "system-name"
                        "logical-host:logical-directory; system-name")
```

`sct:set-system-source-file` *system-name source-file* *Function*

`sct:set-system-source-file` specifies *source-file*, the pathname of the source file that contains the definition of a system called *system-name*.

`sct:set-system-source-file` can be used in one of two ways. The first is recommended.

- When your system is defined with logical pathnames, include the `sct:set-system-source-file` form in the file `sys:site;system-name.system`. `load-system` loads this file the first time you try to load your system. For example, when you type `(load-system 'graphic-lisp)`, `load-system` loads the file `sys: site; graphic-lisp.system`, whose contents are as follows.

```
;;; -*- Mode: LISP; Package: USER -*-
```

```
(fs:make-logical-pathname-host "graphic-lisp")
(sct:set-system-source-file "graphic-lisp"
                            "graphic-lisp: graphic-lisp; glisp-sys")
```

Note that the `sct:set-system-source-file` form must be the second (and last) form in the file, because the logical pathname of the system declaration file (`"graphic-lisp: graphic-lisp; glisp-sys"`) depends on the previous definition of the logical host (`"graphic-lisp"`) in the first form. It is common for the name of the logical host to be the same as the name of the system, as in this example.

- Alternatively, when your system is defined with physical pathnames, you can have your init file evaluate the **sct:set-system-source-file** form (or type the form at a Lisp Listener) prior to calling **load-system** or using one of the relevant Command Processor commands. *source-file* is loaded the first time you compile or load your system.

If a logical host is needed, use the additional form:

```
(fs:make-logical-pathname-host "logical-host")
```

For example, for the system **graphic-lisp** the file `sys: site; graphic-lisp.system` contains the following:

```
;;; -*- Mode: LISP; Package: USER -*-
```

```
(fs:make-logical-pathname-host "graphic-lisp")
(sct:set-system-source-file "graphic-lisp"
  "graphic-lisp: graphic-lisp; glisp-sys")
```

The first form, a call to **fs:make-logical-pathname-host**, defines a logical host. Commonly, the *"logical-host"* is the same name as *"system-name"*. Make sure that the **fs:make-logical-pathname-host** form is the first form in the file, as the second form, (**sct:set-system-source-file** ...), depends on having the logical host defined already. **fs:make-logical-pathname-host** also loads the translations file, which defines the translation from logical pathnames to physical pathnames.

The second form in the *system-name*.file is a call to **sct:set-system-source-file**, which specifies the logical pathname of the system declaration file. **load-system**, after referring to the translation definitions, loads the system declaration file. Note: **si:set-system-source-file** is provided for compatibility with Release 6.1.

### 21.1.2 Sys:site;Logical-host.Translations File

The translations file defines the translation from logical directories on the logical host to physical directories on a physical host. These definitions determine how logical pathnames are translated to physical pathnames. The file contains only one form, a call to **fs:set-logical-pathname-host**, and looks like this.

```
(fs:set-logical-pathname-host "logical-host"
  :physical-host "host-name"
  :translations '(("logical-directory;" "physical-directory")))
```

For example, for the system **graphic-lisp** the file `graphic-lisp.translations` contains the following:

```
;;; -*- Mode: LISP; Package: USER -*-

(fs:set-logical-pathname-host "graphic-lisp"
 :physical-host "waikato"
 :translations '(("graphic-lisp;" ">sys>graphic-lisp>")))
```

To specify a hierarchy of directories instead of a one-to-one translation, you would change the translations list as follows:

```
:translations '(("graphic-lisp;**" ">sys>graphic-lisp>**"))

** means include all subdirectories of "graphic-lisp;".
```

The translations list consists of two-element lists of strings that represent the logical directories specified in the system declaration and their associated physical directories. This list is the only place where your system should refer to a physical host or directory. In simple applications, where all system files are stored in one directory, it is common for the logical directory name (for example, "graphic-lisp;") to be the same as the system name ("graphic-lisp").

This file is loaded in the **file-system** package by the system file, in which the logical host is defined by the function **fs:make-logical-pathname-host**.

### 21.1.3 System Declaration File

This system declaration file contains the **defsystem** form defining your system and, if you need one, the **defpackage** form, which must precede the system declaration. Also this file should contain any user-defined **defsystem** transformations, which must precede the actual system declaration.

Currently, a system declaration file can contain no more than one **defsystem** form, although any number of **defsubsystem** can appear in the file. This is the case because the system declaration can be potentially reloaded for each **defsystem** in the file, causing SCT to become confused.

A sample system declaration file might look like the following:

```
;;; -*- Mode: LISP; Package: CL-USER; -*-
;;; Fortran package specifications
(defpackage fortran-global
  (:use)
  (:nicknames fortran for)
  (:prefix-name "FORTRAN")
  (:colon-mode :external)
  (:size 200))
```

```
(defpackage fortran-system
  (:use)
  (:nicknames for-sys)
  (:prefix-name "FOR-SYS")
  (:colon-mode :external)
  (:size 200))

(defpackage fortran-compiler
  (:use fortran-system fortran-global symbolics-common-lisp)
  (:nicknames for-compiler)
  (:prefix-name "FOR-COMPILER")
  (:colon-mode :external)
  (:size 1500))

(defpackage fortran-user
  (:use fortran-global symbolics-common-lisp)
  (:nicknames for-user)
  (:prefix-name "FOR-USER")
  (:relative-names-for-me (fortran-global user))
  (:size 2000))

;;; System definition using SCT
(defsystem fortran
  (:default-pathname "sys: fortran;"
   :journal-directory "sys: fortran;"
   :patchable t)
  (:module macros ("macros") (:root-module nil))
  (:module language-tools (language-tools) (:type :system))
  (:module front-end (fortran-front-end) (:type :system))
  (:module back-end (fortran-back-end) (:type :system))
  (:serial macros language-tools front-end back-end))

;;; Component system definition
(defsubsystem language-tools
  (:default-pathname "sys: language-tools;")
  (:serial ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-front-end
  (:default-pathname "sys: fortran;")
  (:serial "tokenizer" "grammar" ... ))
```

```
;;; Subsystem definition (non-patchable)
(defsubsystem fortran-back-end
  (:default-pathname "sys: fortran;")
  (:serial "code-generator" "optimizer" ... ))
```

Note the attribute list. The system declaration file is always a Lisp-mode file and is compiled into the `cl-user` package.

The name of the system declaration file does not require an exact format, since you explicitly specify the pathname in the `sct:set-system-source-file` form in the system file. Typically, though, the logical pathname is given as *logical-host:logical-directory;system-name*. The source file should have a canonical file type of `:lisp`. When you call `load-system` the `sct:set-system-source-file` form loads the system declaration file, specifically the `.newest` version.

## 21.2 Loading System Definitions That Use Physical Pathnames

To load system definitions that use physical pathnames, specify the name of the system and the pathname of the system declaration source file in a `sct:set-system-source-file` form. Have your init file evaluate the form (or type the form at a Dynamic Lisp Listener) prior to calling `load-system`.

Note: You are urged to use logical pathnames to ensure that your system is site-independent. A logical pathname has a single translation to a physical pathname. To move your program to another host machine (one perhaps with a different operating system) entails changing only the translation rather than editing all your files to refer to the new file names.





## 22. Other Operations on Systems

Besides being loaded and compiled, systems can be edited, hardcopied, reap-protected, released, and distributed. You can also set the system status and designate a system version. This section describes those operations.

Distribute Systems can be invoked only by a Command Processor command. Setting the system status and version can only be invoked by functions. The other operations can be invoked by either functions or Command Processor commands.

One operation, **si:convert-system-directory**, is needed only for conversion of pre-Genera 7.0 journal files into Genera 7.0 form. See the function **si:convert-system-directory**.

### 22.1 Editing, Hardcopying, Reap-Protecting, and Releasing Systems

**sct:edit-system** *system-name* &key (*query* :confirm) (*silent* nil) *Function*  
 (*batch* nil) (*include-components* t) (*version*  
 :newest) &allow-other-keys

Edit all the source files of the system called *system-name* according to the specified keyword options. This can also be accomplished with the Command Processor command Edit System or the Zmacs command (M-X) Edit System Files.

These are the keyword options to **sct:edit-system**.

**:query** Takes t, nil, :confirm, or :no-confirm. If t, ask for approval of each suboperation, such as whether to load the system declaration file. If nil or :no-confirm, don't ask about anything. If :confirm, list all the suboperations and then ask for confirmation. Default-value: :confirm.

**:silent** Takes t or nil. If t, perform all suboperations without printing anything. If :query is non-nil, :silent t is overridden. Default value: nil.

**:include-components**  
 Takes t or nil. If t, perform the requested system operation on component systems. Default value: t.

**sct:hardcopy-system** *system-name* &key (*query* :confirm) (*silent* nil) (*batch* nil) (*include-components* t) *Function*  
 (*version* :newest) &allow-other-keys

Hardcopies the source files of the system specified by *system-name* according to the specified keyword options.

These are the keyword options to **sct:hardcopy-system**.

**:query** Takes *t*, **nil**, **:confirm**, or **:no-confirm**. If *t*, ask for approval of each suboperation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.

**:silent** Takes *t* or **nil**. If *t*, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.

**:include-components** Takes *t* or **nil**. If *t*, perform the requested system operation on component systems. Default value: *t*.

**sct:reap-protect-system** *system-name* &key (*query* **:confirm**) (*silent* **Function nil**) (*batch* **nil**) (*reap-protect* *t*) (*include-components* *t*) (*version* **:latest**) &allow-other-keys

Reap-protects all the files in the system specified by *system-name* according to the specified options.

These are the keyword options to **sct:reap-protect-system**.

**:query** Takes *t*, **nil**, **:confirm**, or **:no-confirm**. If *t*, ask for approval of each suboperation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.

**:silent** Takes *t* or **nil**. If *t*, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.

**:reap-protect** Takes *t* or **nil**. If *t*, reap-protect the files. If **nil** un-reap-protect them. Default value: *t*.

**:include-components** Takes *t* or **nil**. If *t*, perform the requested system operation on component systems. Default value: *t*.

**sct:set-system-status** *system new-status* &optional *major-version* **Function only-update-on-disk**

Change the status of the specified *system* to *new-status*. Valid values of

*new-status* are: **:experimental**, **:released**, **:frozen**, **:obsolete**, and **:broken**. Note that declaring a system to have a status of **:released** is not the same as designating a system as being the **:released** version. When *only-update-on-disk* is **t**, this does not update in-core datastructures if the system has not been loaded.

**sct:designate-system-version** *system designator major-version* *Function*

*&optional only-update-on-disk*

Add a version designator of *designator* to the specified *major-version* of the *system*. For example, if you want to claim that version 29 of the Tools system is to be called the in-house version:

```
(sct:designate-system-version 'Tools :in-house 29.)
```

When *only-update-on-disk* is **t**, this does not update in-core datastructures if the system has not been loaded.

**sct:release-system** *system major-version &optional* *Function*

*only-update-on-disk*

Puts the **:released** keyword in the *system's* patch directory, inserts a **:released** designation in the system directory. Invoked by the function **sct:release-system**; no corresponding Command Processor command. Releases the system specified by *system-name* with the specified *major-version* number. When *only-update-on-disk* is **t**, this does not update in-core datastructures if the system has not been loaded.

Note: This operation is equivalent to a **sct:set-system-status** operation followed by a **sct:designate-system-version**.

## Distribute Systems Command

Distribute Systems *systems keywords*

Writes systems to tape for distribution. Expects the input to be one or more systems. The default is the first system loaded into the current world. After you confirm the command, the Distribute Systems command lists the systems to write to tape, and asks if you want to perform the operation. Your choices are Y, N, Q, or S. Type Y for Yes, N for No, Q for Quit, or S for Selective. If you choose Selective, each file is listed, and you are asked if you want to load that particular file. You can select as many or as few files as you want. After you enter this information, you are prompted for the name of a tape spec.

*systems* is a list consisting of items separated by commas, each item being either (1) a system name or (2) a system name followed by a space and a version number.

*keywords*           **:File Types** **:Include Component Systems** **:Output Destination**  
                          **:Source Category,** **:Use Disk,** **:Version**

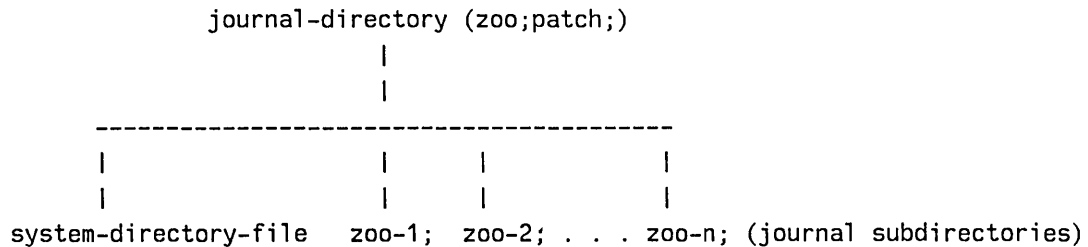
- :File Types        {Sources, Binaries, Both, Patches-Only} What file types to distribute. The default is sources.
- :Include Component Systems  
                  Whether to include the component systems of the systems to be distributed. The default is Yes.
- :Output Destination  
                  {Buffer, File, Printer, Stream} Redirects the output of this command to specified streams.
- :Source Category  
                  {Basic, Optional, Restricted} Indicates which source category or categories to write to tape for distribution. Basic is the default.
- :Use Disk         {Yes, No} Reads the input from disk (test mode), rather than from tape. Test mode writes a special file which is an image of what would be written to tape. Use this when you are preparing a distribution and want to see what files would be written to tape. The default is No.
- :Version         {Latest, Newest, Released} Indicates which version of the system to write to tape for distribution. The type of input expected is: Latest, Newest, or Released. Released is the default.

## 23. Directories Associated with a System

Each system is associated with a set of directories and files. This section explains the directory structure associated with a system called *zoo*. Under a single logical directory, *zoo*; reside these files:

1. System declaration file (contains the **defsystem** form for this system)
2. Multiple versions of source and product files that comprise the system
3. A *journal directory*, by default called *patch*;

The purpose of (1) and (2) should be clear. The journal directory (3), contains a complicated subhierarchy of files. In particular, it contains the *system-directory* file and all of the *journal subdirectories*.



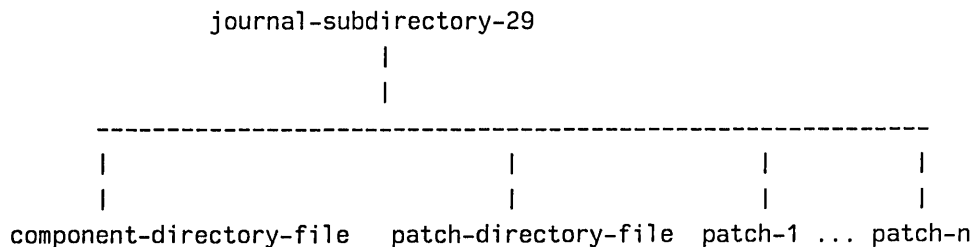
The system-directory file is a registry of the location of the component-directory file for a given version of a system for a given machine. Here is an example of the system-directory file for the *ip-tcp* system.

```

;;; -*- Mode: LISP; Base: 10 -*-
;;; Written 7/05/86 23:01:41 by Zippy,
;;; while running on Brown Creeper from FEP1:>349-1-No-Doc.load.1
;;; ...
(("IP-TCP" :LATEST 51 :SYSTEM-349 51 :SYSTEM-347 49
 :RELEASE-6-G 29 :RELEASE-6-1 29 :RELEASE-6-0 29
 :RELEASE-5-2 23 :RELEASE-5-1 12)
 ;; System versions:
(51
 (:|3600|
 (:COMPONENT-DIRECTORY
 ("SYS:IP-TCP;PATCH;IP-TCP-51.COMPONENT-DIR" :NEWEST NIL))))
(50
 (:|3600|
 (:COMPONENT-DIRECTORY
 ("SYS:IP-TCP;PATCH;IP-TCP-50.COMPONENT-DIR" :NEWEST NIL))))
(49
 (:|3600|
 (:COMPONENT-DIRECTORY
 ("SYS:IP-TCP;PATCH;IP-TCP-49.COMPONENT-DIR" :NEWEST NIL))))
(23
 (:|3600|
 (:COMPONENT-DIRECTORY
 ("SYS:IP-TCP;PATCH;IP-TCP-23.COMPONENT-DIR" :NEWEST NIL))))
(12
 (:|3600|
 (:COMPONENT-DIRECTORY
 ("SYS:IP-TCP;PATCH;IP-TCP-12.COMPONENT-DIR" :NEWEST NIL))))

```

Each journal subdirectory is associated with a particular version of a system. Here is a diagram of the structure of a journal subdirectory for version 29 of a system.



## 23.1 Component Directory File

The *component directory file* is not an actual directory in the file system, but is rather a registry of the source and product version numbers for a major version of a system. Whenever you perform an operation on a system, that operation uses this file to determine the versions of the system files to operate on.

Here is an example of the contents of a component-directory-file.

```
;;; -*- Mode: LISP; Base: 10 -*-
;;; Written 7/05/86 23:01:37 by Zippy,
;;; while running on Brown Creeper from FEP1:>System-349-1.load.1
;;; ...
(("IP-TCP" 51)
 ;; Files for version 51:
 (:|3600|
 (:DEFSYSTEM
 ("SYS:IP-TCP;SYSTEM" 84 NIL))
 (:INPUTS-AND-OUTPUTS
 ("SYS:IP-TCP;TCP-STRUCTURE.TEXT" 10 NIL)
 ("SYS:IP-TCP;TCP-DEFS" 141 92)
 ("SYS:IP-TCP;TCP-ERROR" 34 66)
 ("SYS:IP-TCP;TCP" 265 104)
 ("SYS:IP-TCP;TCP-USER" 119 84)
 ("SYS:IP-TCP;TCP-DEBUG" 57 77)
 ("SYS:IP-TCP;DISTRIBUTION" 66 47)
 .
 .
 .
 ("SYS:IP-TCP;EGP" 83 58))))
```

When you compile a system a new component directory file is created. The major benefit of this detailed record keeping is that your site can support multiple versions of the same system. General users and system developers can load specific versions of systems and specific versions of system files, even when newer and possibly incompatible versions have been made. Some examples:

- System developers can work on the *latest* versions of systems, editing and recompiling some files, without forcing the average user to contend with new and experimental changes to the system.
- General users, on the other hand, can load the stable, *released* versions.
- Symbolics can more easily distribute versions of the system other than the newest version.
- You can use old versions of systems after recompiled versions have been made for the latest system software.



In addition, you can load a system in several different ways:

- by version number
- by version name
- by designation as released, latest, or newest

To load a specific system, use the **:version** option for **load-system**.

The *released* version is the fully debugged version intended for general use. To designate a system as the released version use either **sct:release-system** or **compile-system** with the **:update-directory** option to make the change in the component directory file.

The *latest* version is the most recently compiled version of the system. The component directory file is automatically updated whenever you compile or recompile the system; **compile-system** assigns the **:latest** keyword to this system.

The *newest* version of a system consists of the most recently compiled version of each *file* of a system. The newest version differs from the latest version when individual files have been compiled by hand. The newest version of a system has no version number. Note that you cannot define patches for the newest system.

## 23.2 Contents of the Patch Directory Files

Two *patch-directory* files are created for each patch (one when the patch is begun and another when the patch is finished). The patch-directory file is not a directory; it is a registry of minimum information about a patch including the number of the patch, a comment, the author, and a timestamp. A new patch directory file is created automatically when you recompile a system.

Here is an example of the contents of a patch-directory file.

```

;;; -*- Mode: Lisp; Package: ZL-User; Base: 10.; Patch-File: T -*-
;;; Patch directory for IP-TCP version 51
;;; Written 7/17/86 12:25:42 by Hornig,
;;; while running on Winter from FEP1:>349-19-sc-etc.load.1
;;; ...
(:EXPERIMENTAL
  ((0 "IP-TCP version 51 loaded." "SWM" 2729958587)
   (1
    "Make TFTP work again.
Function TCP::IP-STORE-16: Needs ONCE-ONLY.
Function TCP::IP-STORE-32: ditto
Function (DEFUN-IN-FLAVOR TCP::TFTP-FLUSH-BUFFER
TCP::TFTP-OUTPUT-STREAM):
Recompile caller."
    "Hornig" 2730718516)
   (2
    "Function (DEFUN-IN-FLAVOR TCP::IP-RETRANSMIT-PACKET
TCP::IP-PROTOCOL):
Don't ever forward or redirect broadcast packets."
    "Hornig" 2730990265)
  ))

```

The *patch files* themselves are found in both source and product form, with one source and one product associated with each patch.

Note: In order to convert pre-Genera 7.0 journal files into Genera 7.0 form: See the function `si:convert-journals` in *Converting to Genera 7.0*.



## 24. Patch Facility

Software development is usually a process of incremental changes to many large programs. Many developers can be involved, and the changes can be distributed to any number of users, including the same developers. (Note: the term *large program* refers to one defined by **defsystem**. Only these programs can make use of the patch facility.)

Briefly, developers fix or improve existing functional and other definitions (or write new ones), and then, after thorough testing, decide to issue their changes to the users at their site. They effect release in two ways: (1) they write new versions of the source files containing the edited or new definitions, and (2) they create *patch files*, which contain only the new or changed definitions. Every time a patch is created (written to disk), the patch facility automatically records the event in a sort of "patch registry", noting the number of the patch, the system being patched, and a brief summary of the patch, as described by the developer. Zmacs, the Symbolics editor, provides special tools that make this process relatively easy for the developer.

The patch facility creates a patch file. Saving your buffer after you make a change creates a new version of your source file. When the system is recompiled, your source file, and not the patch file, will be used to construct the new system. The important point is that the patch files – and not the newly written source files – allow the changes to be put into widespread use immediately. The patch facility allows users to obtain all the incremental changes to a system simply by loading its associated patch files.

Basically what occurs during the loading of patches is this: the current state of the patch registry is compared to the registry as last loaded by the user. If patches have been written since that time, just the new patches are loaded, and their summary descriptions are displayed. At that point, the state of the given system in the user's machine is presumably the same as in the developer's machine when the patch was finished.

Genera provides a number of convenient tools and several interfaces for loading patches. For example, users can load patches by calling one of several Lisp functions or alternatively using Command Processor commands. Users also have the choice of loading patches to virtual memory (which means they disappear when the machine is booted) or of saving the patches to disk. (Of course, new patches can be made later, and then these will have to be loaded to get the very latest version of a system.) In the case where users load a particular system whenever they want to use it, the system-loading facility automatically loads all the patches for that system.

Inevitably, a developer or system maintainer must stop accumulating patches and

recompile all the source files in a large program, for example, when a system is changed in a far-reaching way that cannot be accomplished with a patch. Only at this point do the source files become important to system maintenance and distribution. After a complete recompilation, the old patch files are useless and should not be loaded.

To keep track of all the changing number of files in a large program, the patch facility labels each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made; the patch is identified by the major and minor version number together. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number.

The following typical scenario should clarify this scheme.

1. A new system is created; its initial version number is 1.0.
2. Then a patch file is created; the version of the program that results from loading the first patch file into version 1.0 is called 1.1.
3. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2.
4. Then the entire system is recompiled, creating version 2.0 from scratch.
5. Now the two patch files are irrelevant, because they fix old software; the changes that they reflect are integrated into system 2.0.

Note that the second patch file should only be loaded into system 1.1 in order to create system 1.2; you should not load it into 1.0 or any other system besides 1.1. It is important that all the patch files be loaded in the proper order, for two reasons.

- First, it is very useful that any system numbered 1.1 be exactly the same software as any other system numbered 1.1, so that if somebody reports a bug in version 1.1, it is clear just which software is being cited.
- Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

In addition to enabling users to have the most up-to-date programs available, the patch facility performs another important function. Via the patch registry, it allows a site to support multiple versions of the same system. Thus, general users can load a stable, debugged version, while system developers can run the *latest* version of the same system, editing and recompiling files, without forcing the

general user to deal with experimental changes. The detailed record keeping that this capability requires is maintained in a hierarchy of files that is created automatically and updated whenever a system is compiled.

The patch registry also keeps track of all the individual patch files that exist, remembering which version each one creates. A separate numbered sequence of patch files exists for each major version of each system, for example, `lmfs-37-15.lisp`, `lmfs-37-16.lisp`, and so forth. All patches for each major version are stored in the *journal subdirectory* associated with that version of the system. See the section "Directories Associated with a System", page 191.

In addition to the patch files themselves, the *patch-directory file* keeps track of what minor versions exist for a major version. For example, `lmfs-37.patch-dir` contains a listing of the patches made for major version 37, their author, a timestamp, and a comment on why each patch was made.

In order to use the patch facility, you must define your system with `defsystem` and declare it as patchable with the `:patchable` option. (`:patchable` is the default.) When you load your system, it is added to the list of all systems present in the world. Whenever you compile your patchable system, its major version in the file system is incremented; thus a major version is associated with a set of compiled code files.

The patch facility keeps track of which version of each patchable system is present, and where the data about that system reside in the file system. This information can be used to update the Genera world automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to be loaded, ask which patches are already loaded, and add new patches. You can also load patches or whole new systems and then save the entire Genera environment away in a FEP file. See the function `zl:load-and-save-patches`, page 212.

## 24.1 Types of Patch Files

The patch facility maintains several different types of files in the journal subdirectory associated with a major version of your system:

- The patch directory files (two versions for each patch)
- Individual patch files (both source and product versions)

The patch directory file constitutes a sort of "patch registry", recording the number of the patch, the name and version of the system being patched, and a brief description of the patch. One version of the patch directory file is created when starting a patch, and another is created when finishing a patch. (Of course, old versions can be deleted and expunged.) See the section "Component Directory File", page 193.

### 24.1.1 Patch Directory File

The *patch directory file* in the journal subdirectory keeps a listing of the patches (minor versions) that exist for a major version. Each major version of the system has its own patch directory file, which lists the minor version number, any comments about the patch, and the patch author. A new patch directory file is created automatically when you recompile a system.

See the section "Directories Associated with a System", page 191. See the section "Component Directory File", page 193. See the section "Contents of the Patch Directory Files", page 194.

### 24.1.2 Individual Patch Files

Each minor version of the system has a patch source file and a corresponding compiled code file. The individual patch files for a major system version reside in the subdirectory for that major version. (The patch directory file also resides in this subdirectory.) Each patch file is uniquely identified by the major and minor version numbers of the system. For example, `lmfs-37-3.lisp` would be the name of the patch source file for major version #37 and minor version #3 of `lmfs`.

### 24.1.3 Organization of Patch Files

The component directory file, the patch directory file, and the individual patch files are created and maintained automatically, but you will need to know where the patch facility stores these patch files and how to find them on your host.

The patch facility knows which directories to associate with your system by looking at how you specified the `:patchable` option and the `:default-pathname` option in your system declaration. For example, the following `defsystem` declaration will cause the patches to be stored in the logical directory "george: patch;" rather than in the directory that holds the other files of the system, the `pathname default`.

```
:default-pathname "george: george;"  
:patchable t  
:journal-directory "george: patch;"
```

When you do not supply the `journal-directory` then the patches are stored in the directory specified by `:default-pathname`; plus `patch`; In the following example this is the logical directory "george: george; patch;"

```
:default-pathname "george: george;"  
:patchable t
```

The source and compiled code patch files for a major system version are kept in the component directory, along with the component directory file. The patch directory file for a major version resides in this same directory.

#### 24.1.4 Names of Patch Files

The patch facility chooses names for your patch files based on your system definition and on the host.

The host determines the file type and the number of characters in the file name. For example, VMS, UNIX 4.1, and ITS use a computer-generated contraction of the file name. A system directory file name like charlie.system-dir on LMFS would be CHARLI (SDIR) on ITS. Similarly, a patch directory file name like charlie-1.patch-dir on LMFS would be CHA001 (PDIR) on ITS.

The following tables show the physical file types of the system directory file, the patch directory file, and the component directory file for various hosts.

| <i>Host</i> | <i>File type of the system directory file</i> |
|-------------|-----------------------------------------------|
| TOPS-20     | SYSTEM-DIR                                    |
| UNIX 4.1    | sd                                            |
| UNIX 4.2    | system-dir (also sd for compatibility)        |
| VMS         | SPD                                           |
| ITS         | (SDIR)                                        |
| LMFS        | system-dii                                    |
| Multics     | system-dir                                    |

| <i>Host</i> | <i>File type of the patch directory file</i> |
|-------------|----------------------------------------------|
| TOPS-20     | PATCH-DIR                                    |
| UNIX 4.1    | pd                                           |
| UNIX 4.2    | patch-dir (also pd for compatibility)        |
| VMS         | VPD                                          |
| ITS         | (PDIR)                                       |
| LMFS        | patch-dir                                    |
| Multics     | patch-dir                                    |

| <i>Host</i> | <i>File type of the component directory file</i> |
|-------------|--------------------------------------------------|
| TOPS-20     | COMPONENT-DIR                                    |
| UNIX 4.1    | cd                                               |
| UNIX 4.2    | component-dir (also cd for compatibility)        |
| VMS         | CPD                                              |
| ITS         | (CDIR)                                           |
| LMFS        | component-dir                                    |
| Multics     | component-dir                                    |

The format of patch file names varies with the type of file.



- The format of the system directory file is some name chosen by the patch facility followed by the appropriate file type and file version number. For example, the system directory file on LMFS for the **george** system might be:

```
q:>sys>george>patch>george.system-dir.1
```

- The format of the patch directory file name is some name followed by the major version number and the appropriate file type and file version number. For example, the patch directory file on LMFS for major version #38 of **george** might be:

```
q:>sys>george>patch>george-38>george-38.patch-dir.44
```

Note that the file resides in a subdirectory of the same name.

- The format of the individual patch file is some name chosen by the patch facility followed by the major version number, the minor version number, and the appropriate file type and file version number. For example, source patch file #1 for major version #38 of **george** might be:

```
q:>sys>george>patch>george-38>george-38-1.lisp
```

Because the translation rules for generating patch file pathnames are fairly complicated, they are not given here. Instead use the **sct:patch-system-pathname** function to determine the names of your patch files.

**sct:patch-system-pathname** *system type &rest args* *Function*

Given a system name and the type (**:component-directory**, **:system-directory**, **:patch-directory**, **:patch-file**) and additional args required by that type, return the pathname for the file in question. Additional args are, in order, *system-major-version*, *system-minor-version*, and *file-canonical-type*. **:system-directory** requires none of these, **:component-directory** and **:patch-directory** require one, and **:patch-file** all three.

Returns the logical pathname of a patch file. *system* is the name of the system. *type* is **:patch-file**, **:system-directory**, **:component-directory**, **:patch-directory**, or **:patch-file**. Specify also any additional *args* required by the type.

#### **:component-directory**

Returns the logical pathname of the component directory file for the system specified by a major version number, for example:

```
(sct:patch-system-pathname "LMFS"
                           :component-directory 37)
```

The form returns #P"SYS: LMFS; PATCH; LMFS-37.COMPONENT-DIR.NEWEST".

**:system-directory**

Returns the logical pathname of the system directory file for the specified system, for example:

```
(sct:patch-system-pathname "LMFS"
                             :system-directory)
```

The form returns #P"SYS: LMFS; PATCH; LMFS.SYSTEM-DIR.NEWEST".

**:patch-directory**

Supplied with a *major-version-number* argument, it returns the logical pathname of that patch directory file for the given system, for example:

```
(sct:patch-system-pathname "LMFS"
                             :patch-directory 51.)
```

The form returns #P"SYS: LMFS; PATCH; LMFS-51.PATCH-DIR.NEWEST".

**:patch-file** Supplied with the *major-version-number*, *minor-version-number*, and *canonical-type* arguments, it returns the logical pathname of the patch file.

```
(sct:patch-system-pathname "LMFS"
                             :patch-file 51. 2.
                             :lisp)
```

The form returns #P"SYS: LMFS; PATCH; LMFS-51-2.LISP.NEWEST".

To find the physical pathname translation of any of these, send the returned value the **:translated-pathname** message. For example, send the **:translated-pathname** message to the returned value of (sct:patch-system-pathname "LMFS" :system-directory). The form would return #P"q:>sys>lmfs>patch>lmfs.system-dir">.

## 24.2 Making Patches

During a typical maintenance session you might make several changes to existing definitions or write new ones. Rather than recompiling the entire system every time you change a source file, you can copy only the new or revised code into a *patch file* and write the file ("finish" the patch). Whenever you finish a patch, the patch facility automatically compiles the file and records the event in a "patch registry" for the system, noting the number of the patch, the system being patch,

and a brief user-supplied description. As soon as a user loads the patch file (after the system is loaded), the state of the given system in his or her machine is presumably the same as in the developer's machine when the patch was finished.

The patch facility allows you to have several patches in progress at once. Thus you can patch several different systems or several different minor versions of the same system during one work session. The patch facility manages this potentially dangerous situation in the following way. Every time you start a patch, a number and a place in the patch registry is reserved for the patch in production. The patch is marked *in-progress*. When the patch is finished, the entry is completed and the in-progress mark removed. If you decide to abort the patch, the registry entry is automatically deleted.

The ability to have more than one patch in-progress to more than one system makes it imperative that you keep track of the state of your various patches. If a patch is left unfinished (unwritten), the `load-patches` function will load neither the in-progress patch or any subsequent finished patches.

The patch facility considers patches to be active or inactive and in one of the following states: initial, in-progress, aborted, or finished. `Show Patches (m-X)` displays the state of all patches started in this work session. If more than one patch is in progress, one of them is known as the *current patch*. The commands that add patches, like `Add Patch (m-X)`, add only to the patch considered by the patch facility to be the current patch. The command `Select Patch (m-X)` displays a menu of active patches and allows you to make another patch the current one.

In general you should adhere to the following steps in making a patch. It is assumed that your system is patchable; that is, the `:patchable` option appears in the system declaration.

1. You must load (via `load-system`) the major version of the system that you want to patch.
2. Read in the source files you want to edit into a Zmacs buffer. Make all changes and test them thoroughly. Write the source file.
3. Use the appropriate Zmacs commands to make your patch. Begin the patch, using `Start Patch (m-X)`.
4. Add the changed code to the patch buffer by using `Add Patch (m-X)`, `Add Patch Changed Definitions of Buffer (m-X)`, or `Add Patch Changed Definitions (m-X)`.
5. Finish the patch, using `Finish Patch (m-X)`, or abort the patch, using `Abort Patch (m-X)`.

Commands provided for initiating a patch are `Start Patch (m-X)`, `Start Private Patch (m-X)`, and `Add Patch (m-X)`.

### 24.2.1 Start Patch (m-X)

Starts a new patch, prompting you for the name of the system to be patched; it must be a system currently loaded. It assigns a new minor version number for that particular system by writing a new version of the patch directory file with an entry for that minor version number. The patch is marked as in-progress. It starts constructing the patch file in an editor buffer, but does not select the buffer.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. Thus, if two people are patching the same system at the same time, they cannot be assigned the same minor version number.

The command does not actually move any definitions into the patch file. You must explicitly do so with Add Patch Changed Definitions of Buffer (m-X), Add Patch Changed Definitions (m-X), or Add Patch (m-X).

The patch facility permits you to start another patch before finishing the current one. However, if your new patch is to the same system, the patch facility warns you that you already have a patch in progress and allows you to take one of four actions:

- Abort the in-progress patch and start a new patch.
- Finish the in-progress patch and start a new patch.
- Proceed with the second patch (initial patch) for this system and leave the in-progress patch intact.
- Use the existing buffer and do not start a new patch.

### 24.2.2 Start Private Patch (m-X)

Although similar to Start Patch (m-X), Start Private Patch (m-X) does not have any relationship to systems, major and minor version numbers, and official patch directories. Rather it allows you to make a private patch file that you can load, test, and share with other users before you install a numbered patch that is automatically available to all users.

Prompts for a patch note to be saved with the patch. Instead of prompting for a system name, the command prompts for a file name. Start Private Patch does not actually move any definitions into the patch file. Use Add Patch Changed Definitions of Buffer (m-X), Add Patch Changed Definitions (m-X), or Add Patch (m-X) to insert the code. Finishing the patch (using Finish Patch (m-X)) writes it out to the specified file.

Note: Use the Load File command or Load File (m-X) to load a private patch; the Load Patches command and the `load-patches` function do not load private patches.

### 24.2.3 Add Patch (m-X)

Starts a new patch if none is underway, prompts you for a system name, and inserts the region or current definition into the patch buffer. If a patch was in progress, Add Patch (m-X) just adds the region or current definition to the current patch file.

Warns you if your editor buffer conflicts with the system being patched. If you mistakenly use Add Patch on code that does not work, select the buffer containing the patch file and delete it. Then later you can use Add Patch (m-X) on the corrected version. For each patch you add, it queries for a patch comment, which it then inserts in the patch file. Just pressing END means "no comment".

Add Patch (m-X), Add Patch Changed Definitions (m-X), or Add Patch Changed Definitions of Buffer (m-X) insert code into the patch file. If the patch is being made to the system the current buffer's file came from, the commands proceed.

If there is a current patch, and it is not appropriate for the system that the buffer's file came from, you see a menu showing all of the current patches, plus an option to create a new patch appropriate for the buffer, plus an option to abort.

### 24.2.4 Add Patch Changed Definitions of Buffer (m-X)

Add Patch Changed Definitions of Buffer (m-X) selects each definition that was changed in the buffer and asks you whether or not you want the definition patched.

For each definition, you can respond as follows:

| <i>Response</i> | <i>Action</i>                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------|
| Y               | Patches the definition.                                                                                              |
| N               | Skips the definition.                                                                                                |
| P               | Patches the definition and any additional modified definitions in the same buffer without asking any more questions. |

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

For each patch you add, it queries for a patch comment, which it then inserts in the patch file. Just pressing END means "no comment".

### 24.2.5 Add Patch Changed Definitions (m-X)

Add Patch Changed Definitions (m-X) selects a buffer in which definitions were changed and asks whether or not you want to patch the changed definitions. Answering N skips the buffer and proceeds to the next buffer, if any. Answering Y selects each definition that has changed in that buffer and asks you whether or not you want the definition patched. For each definition, you can respond as follows:

| <i>Response</i> | <i>Action</i>                                                                                                                                                   |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Y               | Patches the definition.                                                                                                                                         |
| N               | Skips the definition.                                                                                                                                           |
| P               | Patches the definition and any additional modified definitions in the same buffer without asking any more questions; when done, it proceeds to the next buffer. |

If there are more buffers containing definitions to be patched, it asks questions again when it gets to the next buffer.

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

For each patch you add, it queries for a patch comment, which it then inserts in the patch file. Just pressing END means "no comment".

When making multiple patches during one work session use the Select Patch and Show Patches commands to keep track of patches.

### 24.2.6 Select Patch (m-X)

When you are making more than one patch during a work session, Select Patch (m-X) allows you to choose a different patch as the current patch from a menu of active patches. The patching commands (like Add Patch and Add Patch Changed Definitions of Buffer) insert definitions into the patch file that you have selected as the current patch. To insert patch definitions into another buffer, use Select Patch to choose that buffer as the current patch.

### 24.2.7 Show Patches (m-X)

Show Patches (m-X) displays the state of all patches started in this session. Patches are either active or inactive and can be in one of the following states: initial, in-progress, aborted, or finished. *Inactive patches* are in an aborted or finished state. *Active patches* are in an initial or in-progress state. *Initial* means

that the patch buffer has been initialized but as yet no definitions have been added to the buffer. *In-progress* means that the patch buffer has been initialized and definitions have been added to the buffer.

Show Patches groups the active and inactive patches and identifies the current patch.

After making and testing all of your patches, use the Finish Patch command to install the patch in the system.

#### 24.2.8 Finish Patch (m-X)

Finish Patch (m-X) installs the patch file so that other users can load it. This command saves and compiles the patch file (patches are always compiled). If the compilation produces compiler warnings, the command asks whether or not you want to finish the patch anyway. If you do, or if no warnings are produced, a new version of the patch directory file is written. The in-progress mark is removed from the entry in the patch registry.

The command allows you to edit the patch comments, which are written to the patch directory file. (`load-patches` and `zl:print-system-modifications` print these comments.) It then asks you whether you want to send mail about the patch. If you say "yes", it opens a mail buffer and inserts initial contents, including the name of the patch file and your patch comment.

Note: By default the Finish Patch command queries you about sending mail. You can alter this behavior by changing the value of the variable `zwei:*send-mail-about-patch*`. Its valid values are `:ask`, the default value, which queries the user; `t`, which opens a Zmacs mail buffer without querying; and `nil`, which takes no action regarding the sending of patch mail.

Sometimes you start making a patch file and for a variety of reasons do not finish it – for example, you decide to abort the patch, you need to end your work session at this machine, or your machine crashes. In each of these situations it is of the utmost importance that you leave the patch directory file in a clean state; that is, either go back and finish the patch (as soon as possible!) or deallocate the patch number reserved to you. Failure to do so has unfortunate consequences: users at your site will not be able to load patches.

In your machine has crashed, use Resume Patch (m-X) to reclaim access to the patch number previously assigned to you. You can continue with the patch (assuming you saved the source files just prior to the crash) or use Abort Patch (m-X) to deallocate the patch number. Begin the patch again if you wish. If you simply decide to abandon the patch file, then just use Abort Patch. If you must

boot your machine before finishing the patch, then save the patch buffer and as soon as possible use Resume Patch to read in the relevant patch file; finish the patch or abort it, as you wish.

#### 24.2.9 Abort Patch (m-X)

Abort Patch (m-X) deallocates the minor version number that was assigned by the Start Patch or Add Patch commands. It tells Zmacs that you are no longer interested in making the current patch and offers to kill the patch buffer. The next time you do Add Patch (m-X), Zmacs starts a new patch instead of appending to the one in progress.

#### 24.2.10 Resume Patch (m-X)

Resume Patch (m-X) allows you to return to a patch that you were not able to finish in the same boot session in which you started it; for example, your machine might have crashed or you had to boot your machine suddenly. It reads in the relevant patch file if it was previously saved; otherwise it just reclaims your access to the minor version number allocated to you when you started the patch. Abort or finish the patch.

Under certain circumstances you might find it necessary to recompile and reload a patch file.

#### 24.2.11 Recompile Patch (m-X)

Recompile Patch (m-X) recompiles an existing patch file. This command is useful when, for example, an existing patch needs to be edited or a compiled patch file becomes damaged in some way. Never recompile a patch manually or in any way other than using the Recompile Patch command. This command ensures that source and object files are stored where the patch system can find them.

Use Recompile Patch with caution! Recompiling a patch that has already been loaded by other users can cause divergent world loads.

#### 24.2.12 Reload Patch (m-X)

Reload Patch (m-X) reloads an existing patch file. This command makes it easy to reload a patch file without having to know its pathname.

You might want to have your herald announce private patches that you make. **note-private-patch** adds a private patch to the database in your world and includes the name of the patch in the herald.



**note-private-patch** *string**Function*

Adds a private patch to the database in your world. **note-private-patch** takes a *string* argument. For example, the following adds the private patch called `patch.lisp`:

```
(note-private-patch "s:>smiller>patch.lisp")
```

Subsequent displays of your herald show the inclusion of that patch in your world.

You create private patches using the Start Private Patch (m-X) command and then the standard patch commands for adding to and finishing the patch. Use the Load File command or Load File (m-X) to load a private patch; the load patches command and the **load-patches** function do not load private patches.

### 24.3 Loading Patches

When you command the loading of patches for a software system the current state of the patch registry is compared to the registry as last loaded by the user. If patches have been written since that time, just the new patches are loaded, and their summary descriptions are displayed. As each patch is loaded, the state of the given system in your machine is at the same level as in the developer's machine when he or she finished that particular patch.

The patch registry manages the appropriate loading of patches for a particular system. New patches for a system (since the last loading, if any) are installed until no more remain or until an in-progress patch is encountered. In this last case, loading is halted before the patch in-progress is installed, because the consistency of patches that might follow cannot be guaranteed. The system displays a message indicating the presence of unfinished patches.

Genera provides a number of convenient tools and several interfaces for loading patches. For example, you can load patches by calling one of several Lisp functions – **load-patches** or **zl:load-and-save-patches** – or alternatively, by issuing the Load Patches command in the Command Processor. The effect of these tools differs: **load-patches** and its Command Processor equivalent loads patches to virtual memory, which means they disappear when the machine is booted; **zl:load-and-save-patches** writes the patches to disk. (Of course, new patches can be made later, and then these will have to be loaded to get the very latest version of a system.)

When you call **load-system**, the System Construction Tool automatically loads all the patches for that system, using the same options specified in the call to **load-system**.

## Load Patches Command

Load Patches *system keywords*

Loads patches into the current world for the indicated systems or for all systems. See the function **load-patches**, page 211.

*system*            {All *system-name1*, *system-name2* ... } The default is All.

*keywords*            :Query, :Save, :Show

    :Query            {yes no ask} Whether to ask for confirmation before loading each patch. The default is no.

    :Save             {*file-spec*, Prompt, No-Save} The file in which to save the world with all patches loaded. Omitting this keyword means do not save the world. The default when this keyword is added to your command is Prompt which means save the world and then prompt for a pathname.

    :Show             {yes no ask} Whether to print the patch comments as each patch is loaded. The default is yes.

**load-patches** &optional *systems* &key (*query t*) *silent no-warn*            *Function*  
 Brings the current world up to the latest minor version of whichever major version it is, for all systems present, or for certain specified systems. If there are any patches available, **load-patches** offers to read them. **load-patches** also loads the translations file (*sys:site;logical-host.translations* file) if it has changed. **load-patches** returns *t* if any patches were loaded, and *nil* otherwise.

Note: When you do a **load-system** of a patchable system, **load-system** calls **load-patches** after loading the system. If **load-system** is silent, **load-patches** is silent; if **load-system** asks for confirmation, **load-patches** asks for confirmation.

With no arguments, **load-patches** assumes you want to update all the systems present in this world and asks you whether you want to load each patch.

    :query            Takes *t*, *nil*, :confirm, or :no-confirm. If *t*, ask for approval of each and every operation. If *nil* or :no-confirm, don't ask about anything. If :confirm, list all the operations and then ask for confirmation. Default-value: :confirm.

    :silent           Takes *t* or *nil*. If *t*, perform all operations without printing anything. If :query is non-*nil*, :silent *t* is overridden. Default value: *nil*.

**:no-warn** Takes **t** or **nil**. If **t**, don't bother to print a redefinition warning. Default value: **nil**.

**zl:load-and-save-patches** &rest *keyword-args* *Function*

**zl:load-and-save-patches** first disables network services and MORE processing and then loads any patches that need to be loaded and any new versions of the site files, calling **load-patches** with arguments of **:query nil**.

If no one is logged in, it logs in anonymously. If any patches have been loaded, **zl:load-and-save-patches** prompts for the name of a FEP file in which to save the world load and then calls **zl:disk-save** to actually save the resulting world load. If no patches have been loaded, it restores network services to their state before **zl:load-and-save-patches** was called, and logs out if it has logged in anonymously.

Call **zl:load-and-save-patches** *before* you log in in order to avoid putting the contents of your init file into the saved world load.

Note that loading files asynchronously – particularly patch files – is neither guaranteed to work nor an efficient use of resources. The main process and the background process would compete for resources, and you would lose a lot of time to paging and the scheduler. Furthermore, you cannot expect the correct results from loading patch files in a background process for the following reasons:

- **load-patches** can reset and rebuild the site information.
- When a foreground bug occurs while patches are loading, you cannot determine what system the bug occurred in.
- When you are using a subsystem in the foreground while it is being patched in the background, unexpected problems could arise.
- The file could be doing something that maps over all pathnames, expecting that pathnames would not change while it was running.
- **defflavor** has no locking at load time. Thus, the flavor data structures can be damaged if two processes evaluate **defflavor** simultaneously.

## 25. Obtaining Information About a System

The Command Processor command Show System Definition and the Lisp function **describe-system** are useful means of finding information about a system.

### Show System Definition Command

Show System Definition *system keywords*

Displays a the system definition of *system* including its current patch level, status (experimental or released), and the files that make up the system.

*system*            A loaded system.

*keywords*        :Detailed

:Detailed        {Yes,No} Whether to list all the component systems of the system or not. The default is no, the mentioned default is yes.

**describe-system** *system-name* &key (*show-files* t) *system-op* (*reload*        *Function*  
                  nil) (*recompile* nil) (*version* :latest) *detailed*  
                  &rest *keys* &allow-other-keys

Displays useful information about the system named *system-name*. This includes the name of the system source file, the system package default if any, and component systems. For a patchable system, **describe-system** displays the system version and status, a typical patch file name, the sites maintaining the system, and, if the user wants, a listing of patches.

If **:show-files** is t (the default), it displays the history of the files in the system. Other possible values are nil (do not show file history) and **:ask** (ask the user).

If **:system-op** is t (the default), it displays the operations required to load the system. Other possible values are nil (do not display operations) and **:ask** (ask the user).

If **:reload** is t (the default is nil) the files are reloaded.

If **:recompile** is t (the default is nil) the files are recompiled.

The default version of the system is **:latest**.

The **detailed** argument (t or nil) indicates whether to display the plans for the component systems.

Other useful commands include Show System Modifications and Show System Plan.

## Show System Modifications Command

Show System Modifications *system-name keywords*

With no arguments, Show System Modifications lists all the systems present in this world and, for each system, all the modifications that have been loaded into this world. For each modification it shows the major version number (which will always be the same since a world can only contain one major version), the minor version number, and an explanation of what the modification does, as entered by the person who made it.

If Show System Modifications is called with an argument, only the modifications to *system-name* are listed.

*system-name*        The system for which to show modifications. The default is All.

*keywords*         :Author, :Before, :From, :Matching, :Newest, :Number, :Oldest,  
                      :Since, :Through

:Author            A name. Show modifications by a particular person. For example:

```
                      :show modifications system :author kjones
```

would only show those modifications made by the person whose user ID is kjones.

:Before            A date to serve as one limit for modifications to show:

```
                      :before 11/1/84
```

:From             A number to use as the first modification to show.

:Matching         A string to search for in the comments and only show modifications whose comment contain that string:

```
                      :matching namespace
```

:Newest            A number of modifications to show, for instance the ten most recent ones:

```
                      :newest 10
```

Using this keyword without a number is the same as :newest 1.

:Number            A number. Show only this particular modification. For example:

```
                      Show Modifications :number 6
```

would show modification number 6.

:Oldest            A number of modifications to show, for instance the ten earliest ones:

:oldest 10

Using this keyword without a number is the same as :oldest 1.

:Since A date to serve as one limit for modifications to show.

:Through A number to use as the last modification to show:

:through 17

## Show System Plan Command

Show System Plan *system operation*

Show the system plan (i.e., the order of operations) for the specified *system* under the specified *operation*. *operation* can be All, Compile, Load, Edit, Hardcopy, Reap-protect, or Distribute.

## 25.1 Obtaining Information on System Versions

When a Symbolics computer is booted, it displays a line of information telling you what systems are present, and which version of each system is loaded. This information is returned by the function **sct:system-version-info**. It is followed by a text string containing any additional information that was specified by whoever created the current world load. See the function **zl:disk-save**.

**sct:system-version-info** &optional (*brief-p* nil) *Function*

Returns a string giving information about which systems and what versions of the systems are loaded into the machine (for systems that differ from the released versions) and what microcode version is running. A typical string for it to produce is:

"System 242.264, Zmail 83.42, LMFS 37.31, Vision 10.23, Tape 21.9,  
microcode TMC5-MIC 264, FEP 17"

If *brief-p* is **t**, it uses short names, suppresses the microcode version, any systems that should not appear in the disk label comment, the name System, and the commas:

"242.264 Vis 10.23"

**sct:get-system-version** &optional (*system* "system") *Function*

Returns three values. The first two are the major and minor version numbers of the version of *system* currently loaded into the machine. The third is the status of the system, as a keyword symbol: **:experimental**, **:released**, **:obsolete**, or **:broken**. *system* defaults to **system**. This returns **nil** if that system is not present at all.

Releases have version numbers and status associated with them, just as systems do. Symbolics staff assign the release number.

**sct:get-release-version** *Function*

**sct:get-release-version** returns three values, the release numbers and the status of the current world load:

Major version number

Patch version number or string describing minor patch level

Status of the world load as a keyword symbol:

**:experimental**

**:released**

**:obsolete**

**:broken**

**nil** (when status cannot be determined)

**zl:print-system-modifications** *&rest system-names* *Function*

With no arguments, **zl:print-system-modifications** lists all the systems present in this world and, for each system, all the patches that have been loaded into this world. For each patch it shows the major version number (which will always be the same since a world can only contain one major version), the minor version number, and an explanation of what the patch does, as entered by the person who made the patch.

If **zl:print-system-modifications** is called with arguments, only the modifications to *system-names* are listed.

**sct:patch-loaded-p** *major-version minor-version* *&optional (system "system")* *Function*

A predicate that tells whether the loaded version of *system* is past (or at) the specified patch level. Returns **t** if:

- the major version loaded is *major-version* and the minor version loaded is greater than or equal to *minor-version*
- the major version loaded is greater than *major-version*

Otherwise, the function returns **nil**.

**sct:get-system-input-and-output-source-files** *system* *&optional version* *Function*

Returns a list of pairs of the form (input-file output-file) for a specified *version* of the specified *system*. If *version* is not specified, returns the **:newest** version of the files.

**sct:get-system-input-and-output-defsystem-files** *system* &optional *Function*  
(*version nil*)

Returns the system declaration file for a specified *version* of the specified *system*. If *version* is not specified, returns the **:newest** version of the files.

**sct:get-all-system-input-files** *system* &key (*version nil*) *Function*  
(*include-components nil*)

Return a list of three things: (1) pairs of source and product files, (2) the system declaration files, (3) the input files, for a specified *version* of the specified *system*. With no version returns the **:newest** version of all the files.

**sct:check-system-patch-file-version** &key (*system "system"*) *Function*  
(*major-version*  
(**sct:get-system-version** **sct:system**))  
*minor-version file-version*

Checks to see if the patch file to the system, with the specified *major version* and *minor-version*, of *file-version* has been loaded into the world. If *file-version* is **:none**, checks to see that the patch file has never been loaded. If the check fails, it causes an error. Typically, this form is used in a patch file to ensure that a patch to another system has (or has not) been made.



12 31 86

12 31 86

## **PART V.**

### **Program Counter Metering**



## 26. PC Metering

Program counter (PC) metering is a tool to allow the user to determine where time is being spent in a given program. PC metering produces a histogram that you can interpret to improve the performance of your program.

The mechanism of PC metering is as follows. At regular intervals, the front-end processor (FEP) causes the main processor to task switch to special microcode. This microcode looks up the macro PC that contains the virtual address of the macroinstruction that the processor is currently executing. If this virtual address falls outside the *monitored range*, the microcode increments a count of the number of PCs that missed the monitored range. If the address is within the monitored range, the microcode subtracts the bottom of the monitored range from the PC, leaving a word offset. It then divides the word offset by the number of words per *bucket* and uses that as an index into the *monitor array*. Next, it increments that indexed element of the monitor array. This can only measure statistically where the macro PC is pointing; for the results to be valid, a relatively large number of samples per bucket must be available.

For Symbolics 367X, 365X, 364X, and 363X machines with Rev. 4 of the input/output board (this denotes machines with digital audio), PC metering is performed in the audio microtask and samples at a rate of 50,000 samples per second. This is useful for metering almost all code.

For Symbolics 3600 computers with Rev. 2 of the input/output board, the FEP samples at about 170 samples per second, so PC monitoring is probably valid only for sessions that take longer than five to ten seconds.

You specify a range of the program to be monitored. The range is specified by lower and upper bounding addresses, and compiled functions that lie between those addresses are monitored. The range is divided into some number of buckets. The relative amount of time that the program spends executing in each bucket is measured.

The parameters you specify are the range of addresses to be monitored, the number of buckets, and an array with one word for each bucket.

Some of the metering functions deal with *compiled functions*. In this context a compiled function is either a compiled code object or an **sys:art-16b** array, into which escape functions (small, internal operations used by the microcode) compile.

**meter:make-pc-array** *size*

*Function*

Makes a PC array with *size* number of buckets. This storage is wired, so you probably do not want this to be more than about 64. pages, or (\* 64. **sys:page-size**) words.

**meter:monitor-all-functions** *Function*

Changes the microcode parameters so that the monitor array refers to every possible function in the Lisp world at the time of the execution of **meter:monitor-all-functions**. This usually causes many functions to map into a single bucket, and is therefore useful in obtaining a first estimate of which functions are using a significant portion of the execution time.

**meter:setup-monitor** &optional (*range-start* 0) (*range-end* 268435456) *Function*

Monitors the region between *range-start* and *range-end*.

**meter:monitor-between-functions** *lower-function upper-function* *Function*

Monitors all functions between *lower-function* and *upper-function*. This does not work in some situations, such as:

- You compile a function from a buffer, which puts its definition outside the range
- A previous region is extended, and new functions go there instead of in monotonically increasing virtual addresses.

Example:

```
(defun start-of-library ()())
  ...code...
  (defun end-of-library ()())
  (meter:monitor-between-functions #'start-of-library #'end-of-library)
```

**meter:expand-range** *start-bucket* &optional (*end-bucket start-bucket*) *Function*

Changes the microcode parameters so that the entire monitor array refers only to the functions previously contained within the range specified by *start-bucket* and *end-bucket*. *start-bucket* and *end-bucket* are inclusive bounds.

**meter:report** &optional *function-list* *Function*

Prints a summary of the data collected into the monitor array. You should not have to supply the *function-list* argument.

**meter:start-monitor** &optional (*clear* t) *Function*

Enables collection of PC data. If *clear* is not **nil**, the contents of the monitor array are cleared. If *clear* is **nil**, the array is not modified, so that the new samples are simply added to the old.

**meter:stop-monitor** *Function*

Disables further collection of PC data.

- meter:print-functions-in-bucket** *bucket* *Function*  
Prints all the compiled functions that map into the specified *bucket*.
- meter:list-functions-in-bucket** *bucket* *Function*  
Returns a list of all the compiled functions that map into the specified *bucket*.
- meter:range-of-bucket** *bucket* *Function*  
Returns the virtual address range that maps into the specified *bucket*.
- meter:with-monitoring** *clear body...* *Macro*  
Enables monitoring around the execution of *body*. If *clear* is not **nil**, clears the monitor array first. See the function **meter:start-monitor**, page 222.
- meter:map-over-functions-in-bucket** *bucket function &rest args* *Function*  
Calls *function* for every compiled function in the specified *bucket*. The first argument to *function* should be the compiled function, and any remaining arguments are *args*.
- meter:function-range** *function* *Function*  
Returns two values, the buckets that contain the first and last instructions of *function*.
- meter:function-name-with-escapes** *object* *Function*  
If *object* is a compiled function, returns the function spec of the compiled function. Otherwise, returns **nil**.



## **PART VI.**

# **Program Development Tools and Techniques**

### **Caveat to Program Development Tools and Techniques**

The chapters contained in **Program Development Tools and Techniques** should be read only as a tutorial in work styles for the Genera software environment. The program that is used to illustrate the various tools and techniques presented has not been updated and is not guaranteed to run. The code for the program is in the directory `sys:examples;`.





## 27. Introduction

### 27.1 Purpose

In this chapter we introduce the Symbolics Genera software environment. Using a single example program, we present one style of interacting with that environment in developing Lisp programs. We do not prescribe a "best" style of programming in Genera. Rather, we suggest some techniques and combinations of features that expert Lisp machine programmers advocate. You might find these techniques useful in developing a comfortable and efficient programming style of your own.

### 27.2 Prerequisites

This chapter and the chapters that follow are for you if you will be writing or maintaining Lisp programs and have recently begun to use the Genera software environment on your Symbolics computer. These chapters will be most useful if you have some experience writing Lisp programs and are familiar with basic features of Genera. These chapters are not a survey of Genera facilities, a reference manual, or a Lisp primer. You might find the following Symbolics publications especially helpful:

- See the document *User's Guide to Symbolics Computers*.
- See the section "Getting Help" in *User's Guide to Symbolics Computers*.

### 27.3 Scope

We focus on interaction between programmers and Genera. We present some ways of using Genera features that you might find helpful at each stage of program development. We mention some broad issues of style in designing programs, including modularity and efficiency, but we do not explore program structure in any depth. We do not discuss matters of style in using Lisp, such as appropriate uses for structures and flavors.

This documentation corresponds to the Symbolics 3600-family computers.

### 27.4 Method

We derived the methods we describe here by working with programmers at Symbolics. Some of these programmers were early developers of the Symbolics Lisp Machine itself. Their styles vary. Like most programmers, they generally do not follow a simple textbook sequence of designing, coding, compiling, debugging, recompiling, testing, and debugging again. Instead, they develop programs in repeated cycles, each a sequence of editing, compiling, testing, and debugging. These cycles are often nested. For example, an error in testing a program invokes the Debugger; from the Debugger the programmer types Lisp forms or calls the editor to change and recompile code; an error in retesting code from the Debugger invokes the Debugger again.

## 27.5 Features

Symbolics developers have designed the Genera software environment to accommodate a relatively spontaneous and incremental programming style. Five features make up the integrated programming environment described here.

- *The Zetalisp environment.* The Lisp system code allows you to write programs that are extensions of the environment itself. You can often produce complex programs with comparatively little new code. Zetalisp *flavors* let you build data structures with complex modular combinations of associated procedures and state information.
- *The window system.* Windows permit you to shift rapidly among such activities as editing, evaluating Lisp, and debugging. You can suspend an activity in one window, switch to another, and return to the first without losing its state. You can display several activities on the same screen. Because the window system is itself implemented with Zetalisp flavors, you can modify or create windows for special displays.
- *The Zmacs text editor.* Zmacs has sophisticated means of keeping track of Lisp syntax. It interacts with the Zetalisp environment, letting you find out about existing code and incorporate it into your programs. Unlike some structure editors, Zmacs allows you to leave definitions incomplete until you are ready to evaluate or compile them.
- *Dynamic compiling, linking, and loading.* The compiler is always loaded. You can use single-keystroke commands to compile and load source code from a Zmacs buffer. You can write, compile, test, edit, and recompile code in sections. When you recompile a function definition, the function's callers use the new definition.
- *Interactive debugging.* Errors invoke the Debugger in their dynamic environments. From the Debugger you can examine the stack, change values of variables and arguments, call the editor to change and recompile source code, and reinvoke functions.

## 27.6 Organization

The sequence of steps in developing a program in Genera is too complex to mirror in the linear organization this documentation. We emphasize the cyclical course of program development, but we have organized this documentation in a simple way. We present the main programming sequence in the next three chapters. These deal simply with writing and editing, evaluating and compiling, and debugging code. We discuss particular Zetalisp functions, Zmacs commands, and

other features where they appear most useful or where they present alternatives to common techniques.

The next three chapters require virtually no knowledge of flavors or the window system. But knowing about flavors and windows is essential to advanced use of Genera. For some simple uses of flavors and windows and some programming aids for working with them: See the section "Using Flavors and Windows", page 343.

Throughout, we use as an example the development of a single program that draws the recursive arrows in the cover design for this document. Sandy Schafer and Bernard LaCasse of Schafer/LaCasse created the original design. Richard Bryan of Symbolics wrote and we revised a Lisp program that simulates it. For the complete code: See the section "Calculation Module for the Sample Program", page 383. See the section "Output Module for the Sample Program", page 403.

The code is also in the files SYS: EXAMPLES; ARROW-CALC LISP and SYS: EXAMPLES; ARROW-OUT LISP. (To run the program, load SYS: EXAMPLES; ARROW.) For a reproduction of the design produced on a Symbolics LGP-1 Laser Graphics Printer: See the section "Graphic Output of the Sample Program", page 425.

Many of the techniques and facilities we mention are helpful at more than one stage of program development. Conversely, Genera provides many paths for accomplishing tasks at each stage. As programmers at Symbolics gladly acknowledge, there is more than one way to do almost anything in Genera.

In the chapters that develop the Lisp code for the example program, we use change bars to distinguish new or changed code from code that we have already presented. Whenever we display a line of code that has not appeared before, and whenever we change a line of code that has already appeared, we place a vertical bar (|) next to that line in the left margin. This bar is not part of the code itself. In the following example, we change two lines of the definition of **draw-big-arrow**:

```
(defun draw-big-arrow ()
  ;; Determine coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Determine coordinates of shaft vertexes
    (multiple-value-bind (*p3x* *p3y* *p4x* *p4y*)
      (compute-arrow-shaft-points)
      (draw-big-outline)                ;Outline arrow
      (when *do-the-stripes*
        (stripe-arrowhead))))        ;Stripe head
```



## 28. Writing and Editing Code

Programmers who work the Genera software environment seldom write programs in sequence, from beginning to end, before testing them. They often leave definitions incomplete, skip to other definitions, and then return to finish the incomplete forms. They search for existing code to incorporate into new programs. They edit their work frequently, changing code while writing, testing, and maintaining programs.

In this chapter we discuss Genera features, particularly Zmacs commands and Zetalisp functions, that make this style natural. Many of these features are useful at other stages of programming as well: Editing techniques are important in program maintenance, and methods of learning about existing code are helpful in debugging.

To illustrate programming methods, we develop a program that draws the recursive arrow design that appears on the cover of this book. (The program does not draw the horizontal stripes outside the large arrow.) We produce the figure on a Symbolics LGP-1 Laser Graphics Printer, a Symbolics computer screen, or a file. We develop the program in four stages, beginning with simple procedures to outline the arrows and progressively modifying the code to refine the figure:

1. Drawing the borders of the large arrow and of the smaller recursively drawn arrows that it encloses
2. Drawing the diagonal stripes within the figure, but with uniform thickness and spacing
3. Changing the stripes to vary in thickness and spacing
4. Writing the routines that control the output destination

For the code for the sample program and a reproduction of the LGP image the program produces: See the section "Calculation Module for the Sample Program", page 383. See the section "Output Module for the Sample Program", page 403. See the section "Graphic Output of the Sample Program", page 425.

### 28.1 Using Zmacs

Use the Zmacs text editor to write and edit programs. Zmacs has many features that provide information about Zmacs commands, existing code, buffers, and files. Two features are generally useful: the HELP key and completion. For details: See the section "Getting Help" in *User's Guide to Symbolics Computers*.

---

#### 28.1.1 Using The HELP Key In Zmacs

Pressing the HELP key in a Zmacs editing window gives information about Zmacs commands and variables. The kind of information it displays depends on the key you press after HELP.

---

**Reference**

|                     |                                                                                                                                                         |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| HELP ? or HELP HELP | Displays a summary of HELP options.                                                                                                                     |
| HELP A              | Displays names, key bindings, and brief descriptions of commands whose names or descriptions contain a string you specify. (The A refers to "apropos".) |
| HELP C              | Displays the name and a description of the action of a key you specify by pressing it. (The C refers to "command key".)                                 |
| HELP D              | Displays documentation for a command you specify.                                                                                                       |
| HELP L              | Displays a listing of the last 60 keys you pressed.                                                                                                     |
| HELP U              | Offers to "undo" the last major Zmacs operation, such as sorting or filling, when possible.                                                             |
| HELP V              | Displays the names and values of Zmacs variables whose names contain a string you specify.                                                              |
| HELP W              | Displays the key binding for a command you specify. (The W refers to "where".)                                                                          |
| HELP SPACE          | Repeats the last HELP command.                                                                                                                          |

---

**28.1.2 Zmacs Command Completion**

Some Zmacs operations require you to provide names – for example, names of extended commands, Lisp objects, buffers, or files. You usually supply names by typing characters into a *minibuffer* that appears near the bottom of the screen. Often you do not have to type all the characters of a name; Zmacs offers *completion* over some name spaces. When completion is available, you'll see

**Mouse-R Menu of completions**

in the mouse documentation line.

You can request completion when you have typed enough characters to specify a unique word or name. For extended commands and most other names, completion works on initial

substrings of each word. For example, `m-X com b` is sufficient to specify the extended command `Compile Buffer`. `SPACE`, `COMPLETE`, `RETURN`, and `END` complete names in different ways. `HELP` and [*Zmacs Window (R)*] list possible completions for the characters you have typed.

---

### Reference

|                                         |                                                                                                                                                                                                                                              |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>SPACE</code>                      | Completes words up to the current word.                                                                                                                                                                                                      |
| <code>HELP</code> or <code>c-?</code>   | Displays possible completions in the typeout area. You can click with the mouse to select a completion. Where a possible completion includes ellipsis ( . . . ) you can click either <code>R</code> or <code>L</code> for further expansion. |
| <code>Mouse-R</code>                    | Pops up a menu of possible completions. Select a completion with the mouse.                                                                                                                                                                  |
| <code>COMPLETE</code>                   | Displays the full name if possible.                                                                                                                                                                                                          |
| <code>RETURN</code> or <code>END</code> | Confirms the name if possible, whether or not you have seen the full name.                                                                                                                                                                   |

## 28.2 Preparing to Write Code

When Symbolics programmers begin to write new Lisp programs, they often follow these steps:

1. Enter the Zmacs editor.
2. Create a buffer for a new file for the program.
3. Set the attributes of the buffer and file, including major and minor modes.

---

### 28.2.1 Entering the Editor

Use `SELECT E`, [`Edit`] from the System menu, or the `Select Activity` command to enter Zmacs.



---

**Reference**

|                               |                        |
|-------------------------------|------------------------|
| SELECT E                      | Selects a Zmacs frame. |
| [Edit] (from the System menu) | Selects a Zmacs frame. |
| Select Activity command       | Selects a Zmacs frame. |

---

**28.2.2 Creating a New File**

To store program code in a new file, use Find File (c-X c-F) to create a buffer for the file at the beginning of the editing session. You can then edit the file's attributes or create an attribute list that appears in the text. See the section "Creating a File Attribute List", page 234. You will not have to interrupt later work to name the file or check its attributes before you save it.

---

**Reference**

|                     |                                                                                    |
|---------------------|------------------------------------------------------------------------------------|
| Find File (c-X c-F) | Creates and names a buffer for the file, reading in the file if it already exists. |
|---------------------|------------------------------------------------------------------------------------|

---

**28.2.3 Creating a File Attribute List**

Each buffer and generic pathname has attributes, such as Package and Base, which can also be displayed in the text of the buffer or file as an attribute list. An attribute list must be the first nonblank line of a file, and it must set off the listing of attributes on each side with the characters "-\*.". If this line appears in a file, the attributes it specifies are bound to the values in the attribute list when you read or load the file.

Suppose you want the new program to be part of a package named **graphics** that contains graphics programs. In this case, you want to set the Package attribute to **graphics** in three places: the generic pathname's property list; the buffer data structure; and the buffer text. You can make the change in two ways:

- If the package already exists in your Lisp environment, use Set Package (m-X) to set the package for the buffer. The command asks you whether or not to set the package for the file and attribute list as well. You cannot use this command to create a new package.
- Use Update Attribute List (m-X) to transfer the current buffer attributes to the file and create a text attribute list. Edit the attribute list, changing the package. Use Reparse Attribute List (m-X) to transfer the attributes in the attribute list to the file and

the buffer data structure. If the package you specify by editing the attribute list does not exist in your Lisp environment, Reparse Attribute List asks you whether or not to create it under **global**.

The default value of **zl:base** and **zl:ibase** is 10. If you have been writing code that has a Base attribute in the mode line, you should not experience any difficulties. However, in order to help avoid problems in general, changes have also been made to the editor and compiler:

- In the mode line (the **-\*** line in Lisp source files) are the Base and Syntax attributes. The base can be either 8 or 10 (default). The syntax of a program can be either Zetalisp or Common-Lisp.
- If there is a Base attribute, but no Syntax attribute, the syntax defaults to Common-Lisp.
- If there is a Syntax attribute of Common-Lisp, and no Base attribute, the base is assumed to be 10.
- If there is neither a Base nor a Syntax attribute, Base is assumed to be the default base (10) and the syntax is assumed to be Common-Lisp. Furthermore, a warning is issued to the effect that there is neither a Syntax nor a Base attribute. You should edit your program accordingly. With most programs, the Zmacs command Update Attribute List (**m-X**) will add the appropriate attributes to the mode line, following the above defaults.

When you specify a package by editing the attribute list, you can explicitly name the package's superpackage and, if you want, give an initial estimate of the number of symbols in the package. (If the number of symbols exceeds this estimate, the name space expands automatically.) Instead of typing the name of the package, type a representation of a list of the form (*package superpackage symbol-count*). To indicate that the **graphics** package is inferior to **global** and might contain 1000 symbols, type into the attribute list:

```
Package: (GRAPHICS GLOBAL 1000)
```

For more on file and buffer attributes: See the section "File Attribute Lists" in *Reference Guide to Streams, Files, and I/O*.

---

### Example

Suppose the package for the current buffer is **user** and the base

is 8. We want to create a package called **graphics** for the buffer and associated file. We also want to set the base to 10. If no attribute list exists, we use Update Attribute List (m-X) to create one using the attributes of the current buffer. An attribute list appears as the first line of the buffer:

```
;;; -*- Mode: LISP; Package: USER; Base: 8 -*-
```

Now we edit the buffer attribute list to change the package specification from USER to (GRAPHICS GLOBAL 1000) and to change the base specification from 8 to 10. The text attribute list now appears as follows:

```
;;; -*- Mode: LISP; Package: (GRAPHICS GLOBAL 1000); Base: 10 -*-
```

Finally, we use Reparse Attribute List (m-X). The package becomes **graphics** and the base 10 for the buffer and the file.

---

### Reference

|                              |                                                                                                                                                                                                                                                                                   |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Set <i>attribute</i> (m-X)   | Sets <i>attribute</i> for the current buffer. Queries whether or not to set <i>attribute</i> for the file and in the text attribute list. <i>attribute</i> is one of the following:<br>Backspace, Base, Fonts, Lowercase, Nofill, Package, Patch File, Syntax, Tab Width, or Vsp. |
| Update Attribute List (m-X)  | Assigns attributes of the current buffer to the associated file and the text attribute list.                                                                                                                                                                                      |
| Reparse Attribute List (m-X) | Transfers attributes from the text attribute list to the buffer data structure and the associated file.                                                                                                                                                                           |

---

### 28.2.4 Major and Minor Modes

Each Zmacs buffer has a major mode that determines how Zmacs parses the buffer and how some commands operate. Lisp Mode is

best suited to writing and editing Lisp code. In this major mode, Zmacs parses buffers so that commands to find, compile, and evaluate Lisp code can operate on definitions and other Lisp expressions. Other Zmacs commands, including `LINE`, `TAB`, and comment handlers, treat text according to Lisp syntax rules. See the section "Keeping Track of Lisp Syntax", page 247.

If you name a file with one of the types associated with the canonical type `:lisp`, its buffer automatically enters Lisp Mode. Following are some examples of names of files of canonical type `:lisp`:

| <i>Host system</i> | <i>File name</i>                         |
|--------------------|------------------------------------------|
| Lisp Machine       | acme-blue:>symbolics>examples>arrow.lisp |
| TOPS-20            | acme-20:<symbolics.examples>arrow.lisp   |
| UNIX               | acme-vax:/symbolics/examples/arrow.l     |

You can also specify minor modes, including Electric Shift Lock Mode and Atom Word Mode, that affect alphabetic case and cursor movement. Whether or not you use these modes is a matter of personal preference. If you want Lisp Mode to include these minor modes by default, you can set a special variable in an init file. If you want to exit one of these modes, simply repeat the extended command. The command acts as a toggle switch for the mode.

---

### Example

The following code in an init file makes Lisp Mode include Electric Shift Lock Mode if the buffer's Lowercase attribute is `nil`, as it is by default:

```
(login-forms
  (setf zwei:lisp-mode-hook
        'zwei:electric-shift-lock-if-appropriate))
```

---

### Reference

|                                |                                                                                |
|--------------------------------|--------------------------------------------------------------------------------|
| Lisp Mode (M-X)                | Treats text as Lisp code in parsing buffers and executing some Zmacs commands. |
| Electric Shift Lock Mode (M-X) | Places all text except comments and strings in uppercase.                      |
| Atom Word Mode (M-X)           | Makes Zmacs word-manipulation                                                  |

|                                        |                                                                                                                                                                                                                                                   |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                        | commands (such as <code>m-F</code> ) operate on Lisp symbol names.                                                                                                                                                                                |
| Auto Fill Mode ( <code>m-X</code> )    | Automatically breaks lines that extend beyond a preset fill column.                                                                                                                                                                               |
| Set Fill Column ( <code>c-X F</code> ) | Sets the fill column to be the column that represents the current cursor position. With a numeric argument less than 200, sets the fill column to that many characters. With a larger numeric argument, sets the fill column to that many pixels. |

## 28.3 Program Development: Design and Figure Outline

---

### 28.3.1 Program Strategy

Our goal in developing the sample program is to reproduce the pattern of striped arrows on the cover of this document. The pattern consists of one large arrow enclosing many small arrows that are similar to each other. Each arrow is a series of line segments that form either its outline or its stripes.

We have two general problems in writing the program. We must calculate the position of each line segment we want to draw. We must also convert these positions into a form that will produce line segments on the output device we choose.

In solving these problems, we want to adhere to two principles:

- We want the program to be as modular as possible. The routines that calculate line positions should not depend on the output device we choose. The routines that translate positions for the output device should not depend on any particular method of calculating those positions. If we want to change the internal operation of either set of routines, we should not have to change the other.
- We want to write the program in an incremental style. We write the program in stages, producing a working version at each stage. We start with simple tasks and gradually add refinements until we are satisfied with what the program accomplishes.

We write the program in two modules, one to calculate line

positions and the other to translate positions for the output streams. We put these modules in separate files. For the first file: See the section "Calculation Module for the Sample Program", page 383. For the second file: See the section "Output Module for the Sample Program", page 403.

How do we send line positions from the module that calculates them to the module that transmits them to output? The output module consists of definitions of flavors and methods to transfer information to the appropriate output stream: See the section "Using Flavors and Windows", page 343. Streams for LGP and screen output can both produce lines using the coordinates of the endpoints. Our module that calculates line positions needs to compute the coordinates of the endpoints of the lines to be drawn. In the output module, we define a generic operation called `:show-lines` to receive the coordinates from the calculation module and translate them for the appropriate output stream. The calculation module sends `:show-lines` messages to the output module. We can decide at run time which output stream to use.

Now that we have defined the interface between the two modules, we could in principle write either module first. Although we want the position-calculating routines to be independent of the output device, we have to choose a coordinate system for the calculations. For ease of interpretation, we place the origin at bottom left. This is the convention that the system LGP routines use, but the origin for screen coordinates is at top left. For the sake of convenience, we calculate positions in units of LGP pixels.

---

### 28.3.2 Simple Screen Output

For a discussion of the output routines: See the section "Using Flavors and Windows", page 343. Eventually, we want to produce output on the screen, an LGP, or a file. To develop the program, we need a routine for simple screen display so that we can check the results of our calculation routines. We can use the stream that is the value of `zl:terminal-io`. This stream handles `:draw-line` messages whose arguments include the coordinates of the endpoints of the lines to be drawn. For more on `:draw-line`: See the method (`flavor:method :draw-line tv:graphics-mixin`) in *Programming the User Interface, Volume B*.

We first create a source file for the output routine. We define a flavor, `screen-arrow-output`, and a method to handle `:show-lines` messages from the calculation routines. The arguments to `:show-lines` are the coordinates of the endpoints of one or more lines to be drawn. If the message has more than four arguments

– the coordinates of two endpoints – we assume that we are to draw more than one line, each starting at the endpoint of the last. The `:show-lines` method must iterate over the arguments of the message and send `zl:terminal-io` a `:draw-line` message for each line to be drawn.

We must remember to transform the y-coordinate to take account of the screen's origin at the top. We must also scale both coordinates to take account of the LGP's higher resolution: Screen pixels are about 2.5 times as large as LGP pixels.

The following code provides this simple output module:

```
| (deflavor screen-arrow-output
|   ((scale-factor 2.5))
|   ())
|
| (defmethod (screen-arrow-output :show-lines)
|   (x y &rest x-y-pairs)
|   (loop for x0 = (send self ':compute-x x) then x1
|         for y0 = (send self ':compute-y y) then y1
|         for (x1 y1) on x-y-pairs by #'cddr
|         do (setq x1 (send self ':compute-x x1)
|                y1 (send self ':compute-y y1))
|         (send terminal-io ':draw-line
|                x0 y0 x1 y1 tv:alu-ior t)))
|
| (defmethod (screen-arrow-output :compute-x) (x)
|   (fixr (/ x scale-factor)))
|
| (defmethod (screen-arrow-output :compute-y) (y)
|   (fixr (- 800 (/ y scale-factor))))
```

---

### 28.3.3 Outlining the Figure

We now begin to write the module that calculates the coordinates of the lines that make up the figure. First we must decide how to represent the large arrow that encloses the figure and the smaller arrows inside it. Seven points define each arrow: See the section "Calculation Module for the Sample Program", page 383.

Each arrow has a head, bounded by points 0, 1, and 6, and a shaft, bounded by points 2, 3, 4, and 5. The large outer arrow

and the smaller inner arrows differ in their shafts. Each inner arrow has two yet smaller arrows beneath it. The inferior arrows overlap the shafts of the superior arrows and turn each shaft into a series of descending triangles.

We have two kinds of arrow, represented by the large outer arrow and the small inner ones. We can treat these differences in several ways:

- We can define two structures, make each arrow an instance of one of the structures, and store information about each arrow in the structure's slots. See the section "Structure Macros" in *Symbolics Common Lisp: Language Concepts*.
- We can define two flavors, make each arrow an instance of one of the flavors, and store information about each arrow in the flavor's instance variables. See the section "Flavors" in *Symbolics Common Lisp: Language Concepts*.
- We can simply define global variables to represent the state of the current arrow.

Whichever method we choose, some operations, such as striping the arrowheads, will be the same for both kinds of arrows. Other operations, such as striping the shafts, will depend on the kind of arrow we are drawing.

For simplicity, we use global variables to hold information about the arrows, and we use functions to define procedures for calculating coordinates. Note that we *bind* the global variables rather than *set* them. We do this because we might eventually have two or more arrow programs running at the same time in separate processes. If we set global variables, one program might incorrectly use a value set by another. See the section "The Arrow Window: Interaction, Processes, and the Mouse", page 364.

Our first task in writing the calculation module is to outline the arrows. After creating a file for the module, we write the code for this task in six steps:

1. Define variables to hold information about the arrow we are drawing. For the `:show-lines` message we need the x- and y-coordinates of the seven points that define the arrow. We also need the length of the top edge of the arrow, which we use as a base length. In calculating coordinates, we also need the values of one-half and one-fourth the length of the top edge.

We use `defvar` to declare global variables near the beginning of



the file. This special form declares variables special for the compiler and lets us supply default initial values and documentation strings. By convention, we surround the names of global variables with asterisks to distinguish them from names of local variables.

```
| (defvar *top-edge* nil
|   "Length of the top edge of the arrow")

| (defvar *top-edge-2* nil
|   "Half the length of the top edge")

| (defvar *top-edge-4* nil
|   "One-fourth the length of the top edge")

| (defvar *p0x* nil
|   "X-coordinate of point 0")

| (defvar *p0y* nil
|   "Y-coordinate of point 0")

| (defvar *p1x* nil
|   "X-coordinate of point 1")

| (defvar *p1y* nil
|   "Y-coordinate of point 1")

| (defvar *p2x* nil
|   "X-coordinate of point 2")

| (defvar *p2y* nil
|   "Y-coordinate of point 2")

| (defvar *p3x* nil
|   "X-coordinate of point 3")

| (defvar *p3y* nil
|   "Y-coordinate of point 3")

| (defvar *p4x* nil
|   "X-coordinate of point 4")
```

```

| (defvar *p4y* nil
|   "Y-coordinate of point 4")

| (defvar *p5x* nil
|   "X-coordinate of point 5")

| (defvar *p5y* nil
|   "Y-coordinate of point 5")

| (defvar *p6x* nil
|   "X-coordinate of point 6")

| (defvar *p6y* nil
|   "Y-coordinate of point 6")

```

2. Define an initial function, **draw-arrow-graphic**, for the calculation module. We will call this function from the one we invoke to start the program. We pass **draw-arrow-graphic** the length of the top edge of the large arrow and the coordinates of its top right point (point 0). These arguments determine the position and size of the arrow. The function also calculates the half and quarter lengths of the top edge.

```

| (defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
|   (let ((*top-edge-2* (/ *top-edge* 2))
|         (*top-edge-4* (/ *top-edge* 4))))

```

3. Outline the large arrow. We compute the coordinates of the other six points of the arrow, then send a **:show-lines** message to draw the lines. We can calculate the coordinates of points 1, 2, 5, and 6 the same way for both the large and small arrows. We put these calculations in a separate function so that we can use the same code for both kinds of arrow. We need a constant to hold the destination of the **:show-lines** messages. We must add to **draw-arrow-graphic** a call to **draw-big-arrow**.

```

| (defconst *dest* nil
|   "Destination of :SHOW-LINES messages to output")

```

```

(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4)))
    (draw-big-arrow)))

|
| (defun draw-big-arrow ()
|   (multiple-value-bind
|     (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
|     (compute-arrowhead-points)
|     (multiple-value-bind (*p3x* *p3y* *p4x* *p4y*)
|       (compute-arrow-shaft-points)
|       (draw-big-outline))))

|
| (defun compute-arrowhead-points ()
|   (let* ((p1x (- *p0x* *top-edge*))
|          (p1y *p0y*)
|          (p2x (+ p1x *top-edge-4*))
|          (p2y (- *p0y* *top-edge-4*))
|          (p6x *p0x*)
|          (p6y (- *p0y* *top-edge*))
|          (p5x (- *p0x* *top-edge-4*))
|          (p5y (+ p6y *top-edge-4*)))
|     (values p1x p1y p2x p2y p5x p5y p6x p6y)))

|
| (defun compute-arrow-shaft-points ()
|   (values (- *p1x* *top-edge-4*)
|           (- *p2y* *top-edge-2*)
|           *p2x*
|           (- *p2y* *top-edge*)))

|
| (defun draw-big-outline ()
|   (send *dest* ':show-lines
|         *p0x* *p0y* *p1x* *p1y* *p2x* *p2y* *p3x* *p3y*
|         *p4x* *p4y* *p5x* *p5y* *p6x* *p6y* *p0x* *p0y*))

```

4. Outline the largest of the small arrowheads. We can generate all the interior outlines in the figure by outlining only the heads of the small arrows. We first draw the largest of these arrowheads by analogy with our drawing the large arrow. We can use our function **compute-arrowhead-points** to calculate the coordinates of the vertexes. First we need to halve the value of **\*top-edge\*** and bind new values for the coordinates of the top right point of the arrow.

```

(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4)))
    (draw-big-arrow)
  |   (let ((*top-edge* *top-edge-2*)
  |         (*p0x* (- *p0x* *top-edge-2*)
  |                 (*p0y* (- *p0y* *top-edge-2*)))
  |       (do-arrows))))

|   (defun do-arrows ()
|     (let ((*top-edge-2* (/ *top-edge* 2))
|           (*top-edge-4* (/ *top-edge* 4)))
|         (draw-arrow)))

|   (defun draw-arrow ()
|     (multiple-value-bind
|       (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
|       (compute-arrowhead-points)
|     (draw-outline)))

|   (defun draw-outline ()
|     (send *dest* ':show-lines *p2x* *p2y* *p1x* *p1y*
|           *p0x* *p0y* *p6x* *p6y* *p5x* *p5y*))

```

5. Outline the rest of the small arrows. Each small arrow has two inferior arrows beneath it. We modify our function **do-arrows** by adding two recursive function calls: one to draw the left-hand inferior of each superior arrow, and one to draw the right-hand inferior. We limit the levels of recursion by defining a constant, **\*max-depth\***, and incrementing the variable **\*depth\*** on each call to **do-arrows** until **\*depth\*** equals **\*max-depth\***.

```

|   (defvar *depth* 0
|     "Level of recursion for the current arrow")

|   (defconst *max-depth* 7
|     "Number of levels of recursion")

```

```

(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4)))
    (draw-big-arrow)
    (let ((*top-edge* *top-edge-2*)
          (*p0x* (- *p0x* *top-edge-2*)
                  *p0y* (- *p0y* *top-edge-2*)
                  (*depth* 0))
          (do-arrows))))

(defun do-arrows ()
  (when (< *depth* *max-depth*)
    (let ((*top-edge-2* (/ *top-edge* 2))
          (*top-edge-4* (/ *top-edge* 4)))
      (draw-arrow)
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (+ *top-edge-4* (- *p0x* *top-edge*))
                    *p0y* (- *p0y* *top-edge-4*)))
          (do-arrows))
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (- *p0x* *top-edge-4*)
                    *p0y* (+ *top-edge-4* (- *p0y* *top-edge*)))
          (do-arrows))))))

```

6. Define a function we can call to produce the graphic. This function has to make an instance of **screen-arrow-output**, clear the screen, and call **draw-arrow-graphic**. The arguments to **draw-arrow-graphic** determine the size and placement of the figure. For now, we use estimates based on the dimensions, in pixels, of an LGP page.

```

| (defun do-arrow ()
|   (let ((*dest* (make-instance 'screen-arrow-output)))
|     (send terminal-io ':clear-screen)
|     (draw-arrow-graphic 1280 1800 1800)))

```

We now have a simple working version of our program. We first compile our code: See the section "Compiling Lisp Code", page

298. We then use `SELECT L` to select a Lisp Listener. There we can evaluate `(graphics:do-arrow)` to run the program. We can avoid typing the package prefix by first using `zl:pkg-goto` to make the current package `graphics`:

```
(pkg-goto 'graphics)
```

When we run the program, we generate a screen image of the arrow outlines. Figure 2 shows the output of the program at this stage.

These six steps illustrate a pattern of incremental program development:

- We make each function initially simple. We add new functions and edit old ones as tasks become more complex or refined. Facilities for keeping track of Lisp syntax and for editing code encourage this incremental style. See the section "Keeping Track of Lisp Syntax", page 247. See the section "Editing Code", page 282.
- We compile, test, and debug code in sections as we write it. Many Symbolics programmers, for example, would test `draw-arrow` both before and after adding the recursive function calls.

To support this incremental style, we must be able to check the syntax of our code, edit it, and compile it in sections. See the section "Keeping Track of Lisp Syntax", page 247. See the section "Editing Code", page 282. See the section "Compiling and Evaluating Lisp", page 297.

## 28.4 Keeping Track of Lisp Syntax

Zmacs allows you to move easily through Lisp code and format it in a readable style. Commands for aligning code and features for checking for unbalanced parentheses can help you detect simple syntax errors before compiling.

Zmacs facilities for moving through Lisp code are typically single-keystroke commands with `c-n-` modifiers. For example, Forward Sexp (`c-n-F`) moves forward to the end of a Lisp expression; End Of Definition (`c-n-E`) moves forward to the end of a top-level definition. Most of these commands take arguments specifying the number of Lisp expressions to be manipulated. In Atom Word Mode word-manipulating commands operate on Lisp symbol names; when executed before a name with hyphens, for example, Forward Word (`n-F`) places the cursor at the end of the name rather than before the first hyphen. See the section "Major and Minor Modes", page 236.

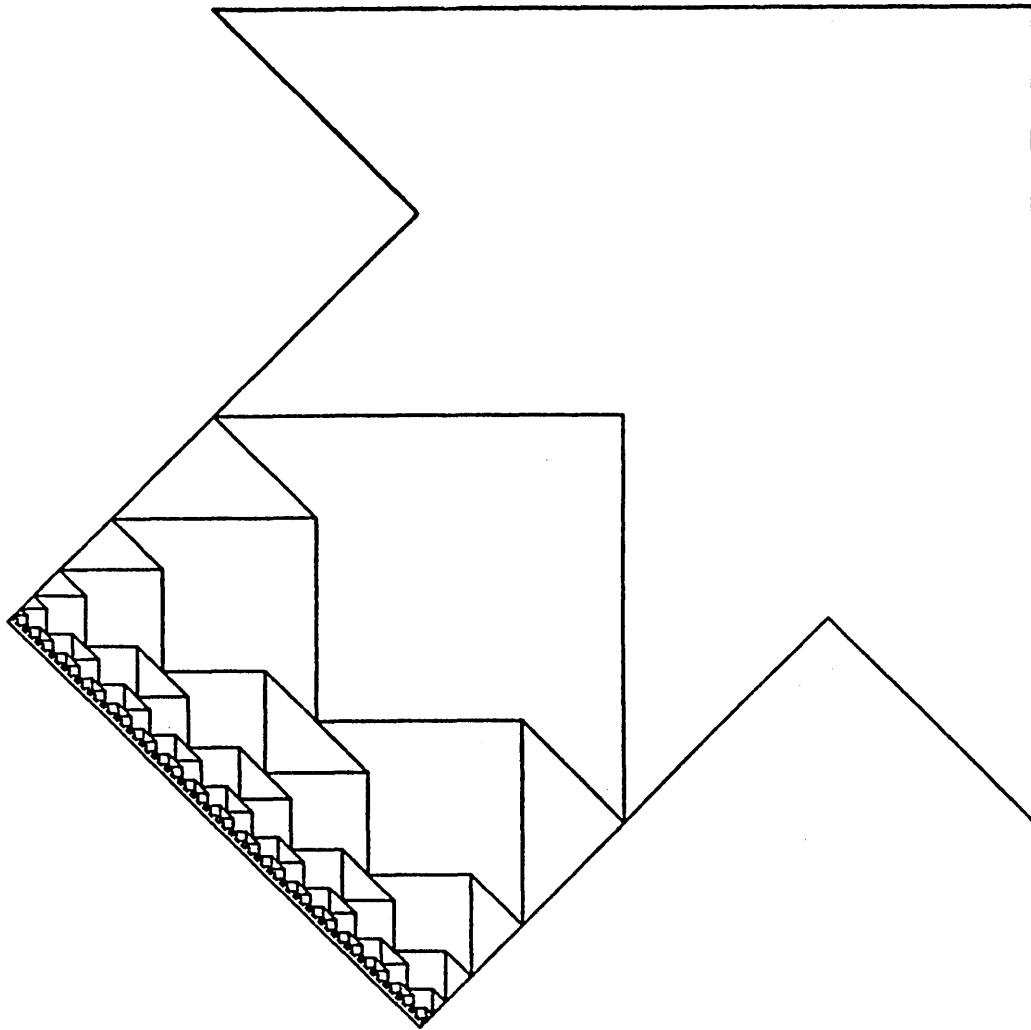


Figure 2. Program output with only the outlines of the arrows in the figure.

For a list of common Zmacs commands for operating on Lisp expressions: See the section "Editing Lisp Programs in Zmacs" in *Text Editing and Processing*.

---

### 28.4.1 Comments

You can document code in two ways. You can supply documentation strings for functions, variables, and constants: See the section "Finding Out About Existing Code", page 260. You can also insert comments in the source code. You can retrieve documentation strings with Zmacs commands and Lisp functions: See the section "Finding Out About Existing Code", page 260. The Lisp reader ignores source-code comments. Although you cannot retrieve them in the same ways as documentation strings, they are essential to maintaining programs and useful in testing and debugging. See the section "Compiling and Evaluating Lisp", page 297. See the section "Debugging Lisp Programs", page 309.

Most source-code comments begin with one or more semicolons. Symbolics programmers follow conventions for aligning comments and determining the number of semicolons that begin them:

- Top-level comments, starting at the left margin, begin with three semicolons.
- Long comments about code within Lisp expressions begin with two semicolons and have the same indentation as the code to which they refer.
- Comments at the ends of lines of code start in a preset column and begin with one semicolon.

`#|` begins a comment for the Lisp reader. The reader ignores everything until the next `|#`, which closes the comment. `#|` and `|#` can be on different lines, and `#|...|#` pairs can be nested.

Use of `#|...|#` always works for the Lisp reader. The editor, however, currently does not understand the reader's interpretation of `#|...|#`. Instead, the editor retains its knowledge of Lisp expressions. Symbols can be named with vertical bars, so the editor (not the reader) behaves as if `#|...|#` is the name of a symbol surrounded by pound signs, instead of a comment.

Now consider `#|...|#`. The reader views this as a comment: the comment prologue is `#|`, the comment body is `|...|`, and the comment epilogue is `|#`. The editor, however, interprets this as a pound sign (`#`), a symbol with a zero length print name (`()`), lisp code (`...`), another symbol with a zero length print name (`()`), and a stray pound sign (`#`). Therefore, inside a `#|...|#`, the editor



commands which operate on Lisp code, such as balancing parentheses and indenting code, work correctly.

---

### Example

Let's add some comments to **draw-arrow-graphic**. We can write a top-level comment without regard for line breaks and then use Fill Long Comment (*m-K*) to fill it. We use *c-* to insert a comment on the current line. We use *m-LINE* to continue a long comment on the next line.

```
;;; This function controls the calculation of the coordinates of the
;;; endpoints of the lines that make up the figure. The three arguments
;;; are the length of the top edge and the coordinates of the top right
;;; point of the large arrow. DRAW-ARROW-GRAPHIC calls DRAW-BIG-ARROW
;;; to draw the large arrow and then calls DO-ARROWS to draw the smaller
;;; ones.
(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4)))
    (draw-big-arrow) ;Draw large arrow
    ;; Length of the top-edge for the first small arrow is half the
    ;; length for the large arrow. Bind new coordinates for the top
    ;; right point of the small arrow.
    (let ((*top-edge* *top-edge-2*)
          (*p0x* (- *p0x* *top-edge-2*))
          (*p0y* (- *p0y* *top-edge-2*))
          (*depth* 0))
      (do-arrows))) ;Draw small arrows
```

---

### Reference

|                                                  |                                                                                          |
|--------------------------------------------------|------------------------------------------------------------------------------------------|
| Indent For Comment ( <i>c-</i> or <i>m-</i> ;) ) | Inserts or aligns a comment on the current line, beginning in the preset comment column. |
| Kill Comment ( <i>c-m-</i> ;) )                  | Removes a comment from the current line.                                                 |
| Down Comment Line ( <i>m-N</i> )                 | Moves to the comment column on the next line. Starts a comment if none is there.         |
| Up Comment Line ( <i>m-P</i> )                   | Moves to the comment column                                                              |

on the previous line. Starts a comment if none is there.

**Indent New Comment Line (m-LINE)**

When executed within a comment, inserts a newline and starts a comment on the next line with the same indentation as the previous line.

**Fill Long Comment (n-X)**

When executed within a comment that begins at the left margin, fills the comment.

**Set Comment Column (c-X ;)**

Sets the column in which comments begin to be the column that represents the current cursor position. With an argument, sets the comment column to the position of the previous comment and then creates or aligns a comment on the current line.

---

### 28.4.2 Aligning Code

Code that you write sequentially will remain properly aligned if you consistently press LINE (instead of RETURN) to add new lines. When you edit code, you might need to realign it. `c-m-Q` and `c-m-\` are useful for aligning definitions and other Lisp expressions.

---

#### Reference

|                                         |                                                                                                                |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <b>Indent New Line (LINE)</b>           | Adds a newline and indents as appropriate for the current level of Lisp structure.                             |
| <b>Indent For Lisp (TAB or c-m-TAB)</b> | Aligns the current line. If the line is blank, indents as appropriate for the current level of Lisp structure. |
| <b>Indent Sexp (c-m-Q)</b>              | Aligns the Lisp expression following the cursor.                                                               |
| <b>Indent Region (c-m-\)</b>            | Aligns the current region.                                                                                     |

---

### 28.4.3 Balancing Parentheses

When the cursor is to the right of a close parenthesis, Zmacs flashes the corresponding open parenthesis. The flashing open parentheses, along with proper indentation, can indicate whether or not parentheses are balanced. Improperly aligned code (after you use a `C-M-Q` command, for instance) is often a sign of unbalanced parentheses.

To check for unbalanced parentheses in an entire buffer, use Find Unbalanced Parentheses (`M-X`). Zmacs can check source files for unbalanced parentheses when you save the files. If a file contains unbalanced parentheses, Zmacs can notify you and ask whether or not to save the file anyway. To put this feature into effect, place the following code in an init file:

```
(login-forms
 (setf zwei:*check-unbalanced-parentheses-when-saving* t))
```

---

#### Reference

Find Unbalanced Parentheses (`M-X`)

Searches the buffer for unbalanced parentheses. Ignores parentheses in comments and strings.

## 28.5 Program Development: Drawing Stripes

So far the sample program outlines all the arrows in the figure. The next task is to draw the diagonal stripes. To keep this stage as simple as possible, we ignore the differences in spacing and thickness of lines in the figure. We draw each stripe from upper left to lower right. We draw the stripes in five steps:

1. Determine the distance between stripes. We first define a constant, `*do-the-stripes*`, that we bind to `t` when we want to draw stripes and `nil` when we want only outlines. We define another constant, `*stripe-distance*`, to contain the horizontal distance between stripes. Let's assume we want 64 stripes in the large arrowhead. We divide the initial `*top-edge*` by 64 to obtain `*stripe-distance*`.

```

| (defconst *do-the-stripes* t
|   "When t, permits striping of the figure")

| (defconst *stripe-distance* nil
|   "Horizontal distance between stripes in the large arrow")

(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4))
        ;; Compute horizontal distance between stripes in the
        ;; large arrow, assuming 64 stripes in the large
        ;; arrowhead.
        (*stripe-distance* (/ *top-edge* 64)))
    (draw-big-arrow) ;Draw large arrow
    ;; Length of the top-edge for the first small arrow is half the
    ;; length for the large arrow. Bind new coordinates for the top
    ;; right point of the small arrow.
    (let ((*top-edge* *top-edge-2*)
          (*p0x* (- *p0x* *top-edge-2*))
          (*p0y* (- *p0y* *top-edge-2*))
          (*depth* 0))
      (do-arrows)))) ;Draw small arrows

```

2. Stripe the head of the large arrow. We define a function, **stripe-arrowhead**, and call it from **draw-big-arrow**. The function loops to calculate the coordinates of the endpoints of the stripes, starting in the upper right corner and decrementing *x* and *y* by **\*stripe-distance\***.

```

(defun draw-big-arrow ()
  ;; Determine coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Determine coordinates of shaft vertexes
    (multiple-value-bind (*p3x* *p3y* *p4x* *p4y*)
      (compute-arrow-shaft-points)
      (draw-big-outline) ;Outline arrow
      (when *do-the-stripes*
        (stripe-arrowhead)))) ;Stripe head

```

```

|   ;;; Function to control striping the head of each arrow.
|   ;;; Determines coordinates of starting and ending points for each
|   ;;; stripe. Calls DRAW-ARROWHEAD-LINES to draw each stripe.
|   (defun stripe-arrowhead ()
|     ;; Find x-coord of top of last stripe to be drawn
|     (loop with last-x = (- *p0x* *top-edge*)
|           ;; Find starting x-coord for each stripe, decrementing
|           ;; by distance between stripes. Stop at last x-coord.
|           for start-x from *p0x* by *stripe-distance* above last-x
|           ;; Find ending y-coord for each stripe, decrementing by
|           ;; distance between stripes.
|           for end-y downfrom *p0y* by *stripe-distance*
|           ;; Draw a stripe
|           do (draw-arrowhead-lines start-x end-y)))

|   ;;; Draws a stripe in an arrowhead. Arguments are the x-coord
|   ;;; of the starting point and the y-coord of the ending point
|   ;;; of a stripe.
|   (defun draw-arrowhead-lines (start-x end-y)
|     (send *dest* ':show-lines start-x *p0y* *p0x* end-y))

```

3. Stripe the exposed portions of the shaft of the large arrow. The shaft consists of a series of descending triangles along the left and right sides. We define a function, **stripe-big-arrow-shaft**, to control the striping. We then define six functions, three to stripe the left side and three to stripe the right. The first function for each side iterates through the triangles that make up the shaft. The second function stripes one triangle. The third function draws one stripe.

```

(defun draw-big-arrow ()
  ;; Determine coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Determine coordinates of shaft vertexes
    (multiple-value-bind (*p3x* *p3y* *p4x* *p4y*)
      (compute-arrow-shaft-points)
      (draw-big-outline) ;Outline arrow
      (when *do-the-stripes*
        (stripe-arrowhead) ;Stripe head
        (stripe-big-arrow-shaft)))) ;Stripe shaft

  |
  |
  |   ;;; Function to control striping the shaft of the large arrow.
  |   ;;; Just calls STRIPE-BIG-ARROW-SHAFT-LEFT to stripe the left side
  |   ;;; and STRIPE-BIG-ARROW-SHAFT-RIGHT to stripe the right side.
  | (defun stripe-big-arrow-shaft ()
  |   (stripe-big-arrow-shaft-left)
  |   (stripe-big-arrow-shaft-right))

  |
  |   ;;; Function to control striping left side of big arrow's shaft.
  |   ;;; Iterates over the triangles that make up the shaft. Determines
  |   ;;; coordinates of the apex and bottom right point of each triangle.
  |   ;;; Calls DRAW-BIG-ARROW-SHAFT-STRIPES-LEFT to stripe each triangle.
  | (defun stripe-big-arrow-shaft-left ()
  |   ;; Set up a counter for depth. Don't exceed maximum recursion
  |   ;; level.
  |   (loop for shaft-depth from 0 below *max-depth*
  |     ;; Find current top edge and its fractions
  |     for top-edge = *top-edge* then (// top-edge 2)
  |     for top-edge-2 = (// top-edge 2)
  |     for top-edge-4 = (// top-edge 4)
  |     ;; Find coordinates of apex of triangle
  |     for apex-x = *p2x* then (- apex-x top-edge-2)
  |     for apex-y = *p2y* then (- apex-y top-edge-2)
  |     ;; Find x-coord of bottom right vertex
  |     for right-x = (+ apex-x top-edge-4)
  |     ;; Find y-coord of bottom edge of triangle
  |     for bottom-y = (- apex-y top-edge-4)
  |     ;; Stripe each triangle
  |     do (draw-big-arrow-shaft-stripes-left
  |       top-edge-4 apex-x apex-y right-x bottom-y)))

```

```
|   ;;; Stripes each triangle in left side of big arrow's shaft.
|   ;;; Arguments are one-fourth current top edge, x- and y-coords
|   ;;; of apex of triangle, x- and y-coords of bottom right vertex.
|   ;;; Determines coordinates of starting and ending points for
|   ;;; each stripe. Calls DRAW-BIG-ARROW-SHAFT-LINES-LEFT to
|   ;;; draw the lines that make up each stripe.
|   (defun draw-big-arrow-shaft-stripes-left
|     (top-edge-4 apex-x apex-y right-x bottom-y)
|     (loop with half-distance = (/ *stripe-distance* 2)
|       ;; Find x-coord of last stripe in triangle
|       with last-x = (- apex-x top-edge-4)
|       ;; Find x-coord of top of each stripe, decrementing
|       ;; from the apex by HALF the horizontal distance
|       ;; between stripes. Stop at last stripe.
|       for start-x from apex-x by half-distance above last-x
|       ;; Find y-coord of top of stripe
|       for start-y downfrom apex-y by half-distance
|       ;; Find x-coord of endpoint of stripe
|       for end-x downfrom right-x by *stripe-distance*
|       ;; Draw a stripe
|       do (draw-big-arrow-shaft-lines-left
|         start-x start-y end-x bottom-y)))

|   ;;; Draws a stripe on the left side of the big arrow's shaft.
|   ;;; Arguments are the coordinates of the starting and ending
|   ;;; points of each stripe.
|   (defun draw-big-arrow-shaft-lines-left
|     (start-x start-y end-x end-y)
|     (send *dest* ':show-lines
|       start-x start-y end-x end-y))
```

```

|   ;;; Function to control striping right side of big arrow's shaft.
|   ;;; Iterates over the triangles that make up the shaft. Determines
|   ;;; coordinates of the top point of each triangle. Calls
|   ;;; DRAW-BIG-ARROW-SHAFT-STRIPES-RIGHT to stripe each triangle.
|   (defun stripe-big-arrow-shaft-right ()
|     ;; Set up a counter for depth. Don't exceed maximum recursion
|     ;; level.
|     (loop for shaft-depth from 0 below *max-depth*
|           ;; Find new top edge and its fractions
|           for top-edge = *top-edge* then (// top-edge 2)
|           for top-edge-2 = (// top-edge 2)
|           for top-edge-4 = (// top-edge 4)
|           ;; Find coords of top point of triangle
|           for start-x = (+ *p2x* top-edge-4)
|           for top-y = (- *p2y* *top-edge-4*)
|           then (- top-y top-edge-2 top-edge-4)
|           ;; Stripe the triangle
|           do (draw-big-arrow-shaft-stripes-right
|               top-edge-2 top-edge-4 start-x top-y)))

|   ;;; Stripes each triangle in right side of big arrow's shaft.
|   ;;; Arguments are one-half and one-fourth of current top edge, and
|   ;;; coords of top point of the triangle. Determines coordinates of
|   ;;; starting and ending points for each stripe. Calls
|   ;;; DRAW-BIG-ARROW-SHAFT-LINES-RIGHT to draw a stripe.
|   (defun draw-big-arrow-shaft-stripes-right
|     (top-edge-2 top-edge-4 start-x top-y)
|     (loop with half-distance = (// *stripe-distance* 2)
|           ;; Find y-coord of last stripe in triangle
|           with last-y = (- top-y top-edge-2)
|           ;; Find y-coord of starting point of stripe. Don't go
|           ;; past the end of the triangle.
|           for start-y from top-y by *stripe-distance* above last-y
|           ;; Find coords of ending point of the stripe, decrementing
|           ;; by HALF the horizontal distance between stripes
|           for end-x downfrom (+ start-x top-edge-4) by half-distance
|           for end-y downfrom (- top-y top-edge-4) by half-distance
|           ;; Draw a stripe
|           do (draw-big-arrow-shaft-lines-right
|               start-x start-y end-x end-y)))

```



```
|
|   ;;; Draws a stripe on the right side of the big arrow's shaft.
|   ;;; Arguments are the coordinates of the starting and ending points
|   ;;; of the stripe.
|   (defun draw-big-arrow-shaft-lines-right
|     (start-x start-y end-x end-y)
|     (send *dest* ':show-lines
|           start-x start-y end-x end-y))
|
```

4. Stripe the heads of the small arrows. We call **stripe-arrowhead** from **draw-arrow**.

```
(defun draw-arrow ()
  ;; Calculate coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    (draw-outline) ;Outline arrowhead
|   (when *do-the-stripes*
|     (stripe-arrowhead))) ;Stripe head
|
```

5. Stripe the exposed shafts of the small arrows. Like the shaft of the large arrow, these shafts are composed of a series of descending triangles. We define three functions: **stripe-arrow-shaft** iterates through the triangles that make up a shaft; **draw-arrow-shaft-stripes** stripes one triangle; and **draw-arrow-shaft-lines** draws one stripe. We call **stripe-arrow-shaft** from **draw-arrow**.

```
(defun draw-arrow ()
  ;; Calculate coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    (draw-outline) ;Outline arrowhead
|   (when *do-the-stripes*
|     (stripe-arrowhead) ;Stripe head
|     (stripe-arrow-shaft))) ;Stripe shaft
|
```

```

|   ;;; Function to control striping the shaft of a small arrow.
|   ;;; Iterates over the descending triangles that make up the shaft.
|   ;;; Calculates the coordinates of the top left and bottom right
|   ;;; vertexes of each triangle.  Calls DRAW-ARROW-SHAFT-STRIPES to
|   ;;; stripe each triangle.
|   (defun stripe-arrow-shaft ()
|     ;; Set up a counter for depth.  Don't exceed maximum
|     ;; recursion level.
|     (loop for shaft-depth from *depth* below *max-depth*
|           ;; Calculate fractions of new top edge
|           for top-edge-2 = *top-edge-2* then (/ top-edge-2 2)
|           for top-edge-4 = (/ top-edge-2 2)
|           ;; Find coords of top left point of triangle
|           for left-x = *p2x* then (- left-x top-edge-4)
|           for top-y = *p2y* then (- top-y top-edge-2 top-edge-4)
|           ;; Find coords of bottom right point of triangle
|           for right-x = (+ left-x top-edge-2)
|           for bottom-y = (- top-y top-edge-2)
|           ;; Stripe the triangle
|           do (draw-arrow-shaft-stripes
|              left-x top-y right-x bottom-y)))

|   ;;; Stripes each triangle in the shaft of a small arrow.
|   ;;; Arguments are coordinates of the top left and bottom
|   ;;; right points of the triangle.  Calculates the y-coord
|   ;;; of the starting point and the x-coord of the ending point
|   ;;; of each stripe.  Calls DRAW-ARROW-SHAFT-LINES to draw the
|   ;;; stripe.
|   (defun draw-arrow-shaft-stripes
|     (left-x top-y right-x bottom-y)
|     ;; Find y-coord of starting point of stripe.  Don't go
|     ;; below the bottom of the triangle.
|     (loop for start-y from top-y by *stripe-distance* above bottom-y
|           ;; Find x-coord of ending point of the stripe
|           for end-x downfrom right-x by *stripe-distance*
|           ;; Draw a stripe
|           do (draw-arrow-shaft-lines
|              left-x start-y end-x bottom-y)))

```

```
|   ;;; Draws a stripe in the shaft of a small arrow. Arguments are  
|   ;;; the coordinates of the starting and ending points of the  
|   ;;; stripe.  
|   (defun draw-arrow-shaft-lines  
|     (left-x start-y end-x bottom-y)  
|     (send *dest* ':show-lines  
|         left-x start-y end-x bottom-y))
```

Figure 3 shows the output of the program, with stripes of even spacing and thickness.

This stage in program development differs from the beginning of the program in two ways:

- As we add new functions, we need to refer to existing code for such information as the order of arguments in argument lists and the values of variables and constants. See the section "Finding Out About Existing Code", page 260.
- We must start to change existing code, adding function calls and new arguments. These changes require increasing use of facilities for editing code. See the section "Editing Code", page 282.

## 28.6 Finding Out About Existing Code

When you write or edit programs, you often need to find characteristics of existing code. If you write programs incrementally, you need to find existing definitions, argument lists, and values. To maintain modularity, you must know how new code should interact with previously written modules. If you want to incorporate parts of the Symbolics system in your programs, you often have to refer to system source code.

Zmacs and Zetalisp have many facilities for retrieving information about Lisp objects and for displaying and editing source code. This section describes features especially useful for writing and editing code. We discuss facilities for learning about Lisp objects, symbols, variables, functions, and pathnames.

---

### 28.6.1 Finding Out About Objects

**describe** displays information about a Lisp object in a form that depends on the object's type. For example, for a special variable, **describe** displays the value, package, and properties, including documentation, pathname of the source file, and Zmacs buffer sectioning node.

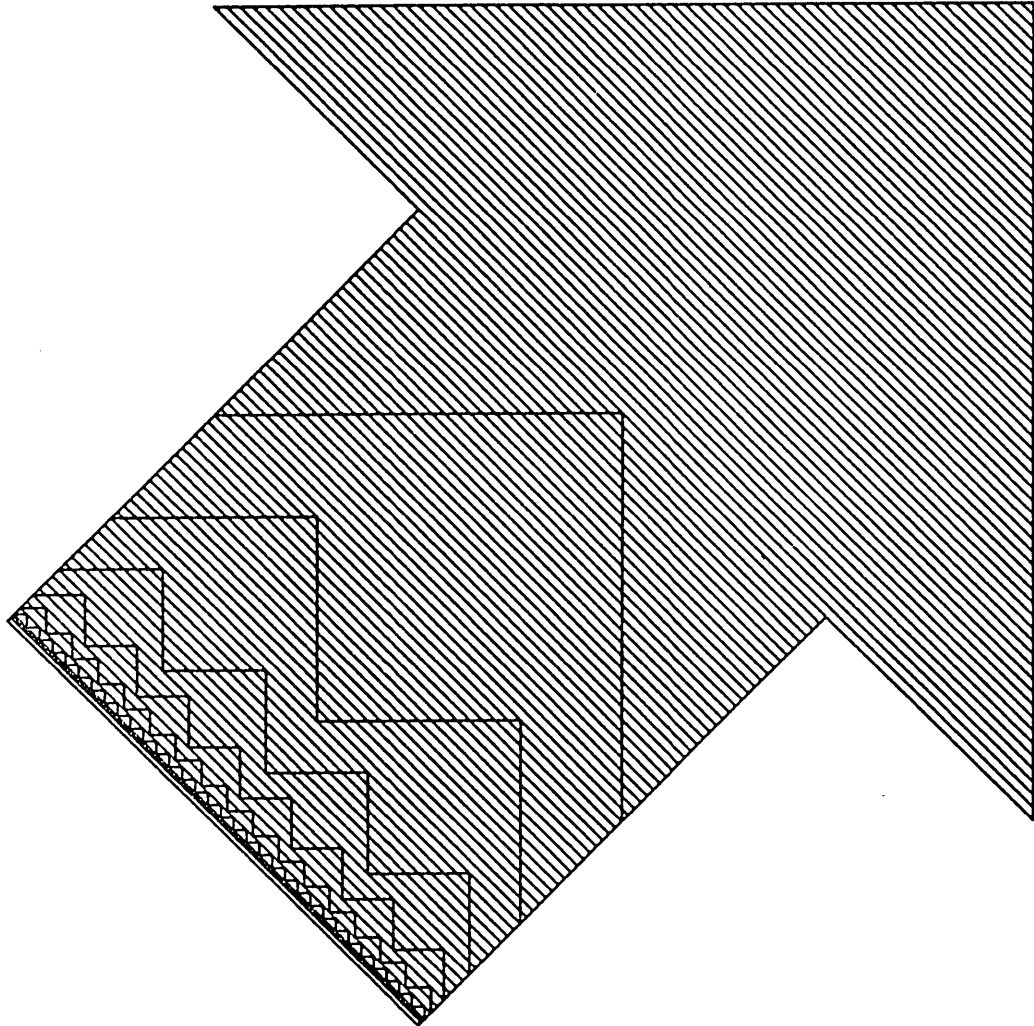


Figure 3. Program output with stripes of even spacing and density.

An interactive, window-oriented version of **describe** is the Inspector. See the section "Using the Inspector", page 335.

**describe** does not display array elements. For that you can use the Inspector or **zl:listarray**.

---

### Example

```
(describe '*top-edge*)
```

The value of \*TOP-EDGE\* is NIL

\*TOP-EDGE\* is in the GRAPHICS package.

\*TOP-EDGE\* has property DOCUMENTATION:

"Length of the top edge of the arrow"

\*TOP-EDGE\* has property SPECIAL:

#<UNIX-PATHNAME "VIXEN: //dess//doc//workstyles//pcodex.↔">

#<UNIX-PATHNAME "VIXEN: //dess//doc//workstyles//pcodex.↔">,
 an object of flavor FS:UNIX-PATHNAME,

has instance variable values:

FS:HOST: #<UNIX-CHAOS-HOST SCRC-VIXEN>

FS:DEVICE: :UNSPECIFIC

FS:DIRECTORY: ("desś" "doc" "workstyles")

FS:NAME: "pcodex"

FS:TYPE: NIL

FS:VERSION: :UNSPECIFIC

SI:PROPERTY-LIST: (BASE 10 :MODE ...)

FS:STRING-FOR-PRINTING: "VIXEN: //dess//doc//workstyles//pcodex.↔"

FS:STRING-FOR-HOST: "//dess//doc//workstyles//pcodex.↔"

FS:STRING-FOR-EDITOR: NIL

FS:STRING-FOR-DIRED: NIL

FS:STRING-FOR-DIRECTORY: NIL

\*TOP-EDGE\* has property SOURCE-FILE-NAME:

((DEFVAR #<UNIX-PATHNAME

"VIXEN: //dess//doc//workstyles//pcodex.↔">))

((DEFVAR #<UNIX-PATHNAME

"VIXEN: //dess//doc//workstyles//pcodex.↔">)) is a list

\*TOP-EDGE\* has property ZWEI:ZMACS-BUFFERS:

((DEFVAR #<SECTION-NODE Variable \*TOP-EDGE\* 27316607>))

((DEFVAR #<SECTION-NODE Variable \*TOP-EDGE\* 27316607>)) is a list

\*TOP-EDGE\*

---

**Reference****(describe *object*)**

Displays information about *object* in a form that depends on the object's type. For named structures, displays the symbolic names and contents of the entries in the structure.

**(listarray *array*)**

Returns a list whose elements are the elements of *array*.

---

**28.6.2 Finding Out About Symbols**

Several Zmacs commands and Lisp functions find the name of a symbol or retrieve information about it. Unless you specify a package, most of these commands search the **global** package and its inferiors. It now takes several minutes to search all these packages; if you don't know which one the symbol is in, you might want to use functions like **zl:apropos** and **who-calls** only as a last resort. For more on the meanings and default values of arguments to these functions: See the section "Getting Help" in *User's Guide to Symbolics Computers*.

---

**Example**

In defining the function **stripe-big-arrow-shaft-left**, we need to use the constant **\*max-depth\***, but we remember only that its name contains "depth". We use either `M-ESCAPE` (to evaluate a form in the editor minibuffer) or `SELECT L` (to select a Lisp Listener) and then evaluate:

```
(apropos "depth" 'graphics)

GRAPHICS:DEPTH
GRAPHICS:*MAX-DEPTH* - Bound
GRAPHICS:SHAFT-DEPTH
GRAPHICS:*DEPTH* - Bound
(*DEPTH* SHAFT-DEPTH *MAX-DEPTH* DEPTH)
```

---

**Example**

After compiling **stripe-arrowhead** we want to test the program as

written so far, but we forget which function calls **draw-arrow-graphic**:

(who-calls 'draw-arrow-graphic 'graphics)

DO-ARROW calls DRAW-ARROW-GRAPHIC as a function.  
(DO-ARROW)

You can also find the callers of a function with List Callers (m-X). See the section "Finding Out About Functions", page 265.

---

## Reference

**(zl:apropos *string package inferiors superiors*)**

Displays the names of all symbols whose names contain *string*. Indicates whether or not the symbol is bound. Displays argument lists of functions.

**Where Is Symbol (m-X)**

Displays the names of packages that contain the specified symbol.

**(where-is *string package*)**

Displays the names of packages that contain a symbol whose print name is *string*.

**(who-calls *symbol package inferiors superiors*)**

Displays information about uses of *symbol* as function, variable, or constant. Returns a list of the names of callers of *symbol*.

**(what-files-call *symbol package*)**

Displays names of files that contain uses of *symbol* as function, variable, or constant.

**(zl:plist *symbol*)**

Returns the list representing the property list of *symbol*.

**List Matching Symbols (m-X)**

Displays the names of symbols for which a predicate lambda-expression returns something other than nil. Prompts for a predicate for the expression

(**lambda** (symbol) *predicate*).  
 By default, searches the current package; with an argument of `c-U`, searches all packages; with an argument of `c-U c-U`, prompts for the name of a package. Press `c-.` to edit definitions of symbols that satisfy the predicate.

---

### 28.6.3 Finding Out About Variables

**Describe Variable At Point** (`c-sh-V`) is a useful command to display information about a variable. It tells you whether or not the variable is bound, whether it has been declared special, and the file, if any, that contains the declaration. You can find the value of a variable by evaluating it in a Lisp Listener. If you have added a documentation string to the variable declaration, you can retrieve the string with `c-sh-V` or with `c-sh-D`, `m-sh-D`, or **documentation**. See the section "Finding Out About Functions", page 265.

---

#### Example

In writing **stripe-arrow-shaft** we want to find out whether or not **\*max-depth\*** is bound. `c-sh-V` displays the following information:

```
*MAX-DEPTH* has a value and is declared special by file
VIXEN: /doss/doc/workstyles/pcodex.l
Number of levels of recursion
```

---

#### Reference

**Describe Variable At Point** (`c-sh-V`)

Indicates whether or not the variable is declared special, is bound, or is documented by **defvar** or **zl:defconst**.

---

### 28.6.4 Finding Out About Functions

Many Zmacs and Zetalisp facilities for finding out about functions apply both to functions defined by **defun** and to objects defined by other special forms and macros that begin with "def".



### 28.6.4.1 Definitions

Edit Definition (`m-.`) is a powerful command to find and edit definitions of functions and other objects. It is particularly valuable for finding source code, including system code, that is stored in a file other than that associated with the current buffer. It finds multiple definitions when, for example, a symbol is defined as a function, a variable, and a flavor. It maintains a list of these definitions in a support buffer, where you can use `m-.` to return to the definitions even when you are finished editing.

For a description of how to use Edit Definition (`m-.`) to edit definitions of flavor methods: See the section "Methods", page 378.

---

#### Example

We have written `stripe-arrowhead` and want to call it from `draw-big-arrow`. We use `m-.` to position the cursor at the definition of `draw-big-arrow`.

---

#### Reference

|                                      |                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Edit Definition ( <code>m-.</code> ) | Selects a buffer containing a function definition, reading in the source file if necessary. You can specify a definition by typing the name into the minibuffer or clicking on a name already in the buffer. Offers name completion for definitions already in buffers. With a numeric argument, selects the next definition satisfying the most recently specified name. |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 28.6.4.2 Names

Often you know only part of a function name and need to find the complete name. Use Function Apropos (`m- $\times$` ).

---

#### Example

We want to call `stripe-arrowhead` from `draw-arrow`, but we remember only that `draw-arrow` contains the string "arrow". We use Function Apropos (`m- $\times$` ) to display the names of functions that contain "arrow". We click left on the name `draw-arrow` to edit its definition.

m-X Function Apropos arrow

Functions matching arrow:

DO-ARROW  
DO-ARROWS  
DRAW-ARROW  
DRAW-ARROW-GRAPHIC  
DRAW-ARROWHEAD-LINES  
DRAW-BIG-ARROW  
DRAW-BIG-ARROW-SHAFT-LINES-LEFT  
DRAW-BIG-ARROW-SHAFT-LINES-RIGHT  
DRAW-BIG-ARROW-SHAFT-STRIPES-LEFT  
DRAW-BIG-ARROW-SHAFT-STRIPES-RIGHT  
STRIPE-ARROWHEAD  
STRIPE-BIG-ARROW-SHAFT  
STRIPE-BIG-ARROW-SHAFT-LEFT  
STRIPE-BIG-ARROW-SHAFT-RIGHT

---

### Reference

|                        |                                                                                                                                                                      |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Function Apropos (m-X) | Displays the names of functions that contain a string. Press <code>c-.</code> or click left on names in the display to edit the definitions of the functions listed. |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 28.6.4.3 Documentation Strings

Function definitions can include documentation strings. When you need to know the purpose of the function, you can retrieve the documentation with `c-sh-D`, `n-sh-D`, or **documentation**.

---

### Example

We wrote a long source-code comment at the beginning of the definition of **draw-arrow-graphic**. We could have made this comment a documentation string:

```

(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  "Function controlling the calculation module.
  Controls calculation of the coordinates of the endpoints of the lines
  that make up the figure. The three arguments are the length of the top
  edge and the coordinates of the top right point of the large arrow.
  DRAW-ARROW-GRAPHIC calls DRAW-BIG-ARROW to draw the large arrow and then
  calls DO-ARROWS to draw the smaller ones."
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4))
        ;; Compute horizontal distance between stripes in the
        ;; large arrow, assuming 64 stripes in the large
        ;; arrowhead.
        (*stripe-distance* (/ *top-edge* 64)))
    (draw-big-arrow) ;Draw large arrow
    ;; Length of the top-edge for the first small arrow is half the
    ;; length for the large arrow. Bind new coordinates for the top
    ;; right point of the small arrow.
    (let ((*top-edge* *top-edge-2*)
          (*p0x* (- *p0x* *top-edge-2*))
          (*p0y* (- *p0y* *top-edge-2*))
          (*depth* 0))
      (do-arrows))) ;Draw small arrows

```

Later, when defining **do-arrow**, we add a call to **draw-arrow-graphic**. We want to be sure that this is the control function for the calculation module. We position the cursor at the name **draw-arrow-graphic** inside the definition of **do-arrow** and use **m-sh-D** to display the documentation for **draw-arrow-graphic**:

```

DRAW-ARROW-GRAPHIC: (*TOP-EDGE* *P0X* *P0Y*)
Function controlling the calculation module.
Controls calculation of the coordinates of the endpoints of the lines
that make up the figure. The three arguments are the length of the top
edge and the coordinates of the top right point of the large arrow.
DRAW-ARROW-GRAPHIC calls DRAW-BIG-ARROW to draw the large arrow and then
calls DO-ARROWS to draw the smaller ones.

```

**c-sh-D** displays the summary documentation:

DRAW-ARROW-GRAPHIC: Function controlling the calculation module.

---

### Reference

|                             |                                               |
|-----------------------------|-----------------------------------------------|
| Show Documentation (m-sh-D) | Displays the function's documentation.        |
| Long Documentation (c-sh-D) | Displays the function's documentation string. |
| (documentation function)    | Displays the function's documentation string. |

#### 28.6.4.4 Argument Lists

Quick Arglist (c-sh-R) and **arglist** retrieve the argument list for an ordinary function, a generic function, or a **send** form with a constant message name. What these facilities display depends on the nature of the function, whether or not it has been compiled, and what options the function includes. For details: See the function **arglist** in *Symbolics Common Lisp: Language Dictionary*. See the section "Getting Help" in *User's Guide to Symbolics Computers*.

---

### Example

We are editing the definition of **do-arrow** to add a call to **draw-arrow-graphic**. We want to see the argument list for **draw-arrow-graphic**. We position the cursor at the name **draw-arrow-graphic** in the definition of **do-arrow** and use c-sh-R:

```
DRAW-ARROW-GRAPHIC: (*TOP-EDGE* *P0X* *P0Y*)
```

---

### Reference

|                        |                                                                                                                                                                                                       |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Quick Arglist (c-sh-R) | Displays a representation of the argument list of the current function. With a numeric argument, you can type the name of the function into the minibuffer or click on a function name in the buffer. |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

(*arglist function*)                      Displays a representation of the function's argument list.

#### 28.6.4.5 Callers

When you change a function definition, you sometimes need to make complementary changes in the function's callers. Four Zmacs commands find the callers of a function. These commands, like **who-calls**, now take several minutes to search all packages for callers. (For the example program, we need to search only the **graphics** package.) By default, these commands search the current package. With an argument of **c-U**, they search all packages. You can specify the packages to be searched by giving the commands an argument of **c-U c-U**.

---

#### Example

We decide to change the order of the arguments to **draw-arrow-graphic**. We want to be sure to change all the callers of **draw-arrow-graphic** to call the function with arguments in the correct order. We use Edit Callers (**m-X**).

---

#### Reference

|                                      |                                                                                                                                                                                          |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| List Callers ( <b>m-X</b> )          | Lists functions that call the specified function. Press <b>c-.</b> to edit the definitions of the functions listed.                                                                      |
| Multiple List Callers ( <b>m-X</b> ) | Lists functions that call the specified functions. Continues prompting for function names until you press only RETURN. Press <b>c-.</b> to edit the definitions of the functions listed. |
| Edit Callers ( <b>m-X</b> )          | Prepares for editing the definitions of functions that call the specified function. Press <b>c-.</b> to edit subsequent definitions.                                                     |
| Multiple Edit Callers ( <b>m-X</b> ) | Prepares for editing the definitions of functions that call the specified functions. Continues prompting for function names until you press only                                         |

RETURN. Press `c-.` to edit subsequent definitions.

---

### 28.6.5 Finding Out About Pathnames

Zmacs provides several ways of finding the name of a file. If you just need the name of a file and have some idea what directory it is in, you can use `c-X c-D` with an argument of `c-U` or View Directory (`m-X`) to display a directory. If you want to operate on files in a directory, you can use `c-X D` with an argument of `c-U` or Dired (`m-X`) to edit a directory. If you want to find a source file but don't know what directory it is in, you might remember the name of a function defined in the file. In that case, you might be able to use `m-.` to find the file.

---

#### Example

After editing the definitions in the calculation module, we want to find the output module to edit the definition of `do-arrow`. We forget the name of the file, but we remember the name of the directory. We can use `c-U c-X c-D` to display the directory. If we have interned `do-arrow` or read its file into a buffer, we can use `m-.` to find `do-arrow` directly.

---

#### Reference

|                                            |                                                                                                                                                                                                                          |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Display Directory ( <code>c-X c-D</code> ) | Displays the current buffer's file's directory. With an argument of <code>c-U</code> , prompts for a directory to display.                                                                                               |
| View Directory ( <code>m-X</code> )        | Lists a directory.                                                                                                                                                                                                       |
| REFRESHr Dired ( <code>c-X D</code> )      | Edits the current buffer's file's directory. With an argument of <code>c-U</code> , prompts for a directory to edit. Displays the files in the directory. You can use single-character commands to operate on the files. |
| Dired ( <code>m-X</code> )                 | Edits a directory. Displays the files in the directory. You can use single-character commands to operate on the files.                                                                                                   |

## 28.7 Program Development: Refining Stripe Density and Spacing

At this stage of development, the program outlines the arrows in the figure and fills them with stripes of uniform thickness and spacing. In the finished figure, stripe thickness or density increases from upper right to lower left within each arrow, and stripe spacing varies among the levels of the figure. We adjust the stripe spacing by replacing the constant distance between stripes by a variable. We correct the stripe density by drawing multiple adjacent lines for each stripe.

We adjust the stripe spacing in three steps:

1. Define a variable, **\*stripe-d\***, to represent the distance between stripes for each arrow.

```
| (defvar *stripe-d* nil
|   "Horizontal distance between stripes for each arrow")
```

2. Calculate the value of **\*stripe-d\*** for each arrow. For the large arrow, this is just **\*stripe-distance\***. For the small arrows, we need to call a new function, **compute-stripe-d**, from **draw-arrow**. **compute-stripe-d** calculates **\*stripe-d\*** as a fraction of **\*stripe-distance\*** that depends on the level of recursion. It ensures that **\*stripe-d\*** divides **\*top-edge\*** evenly and that **\*stripe-d\*** is never less than 3.

```
(defun draw-big-arrow ()
  ;; Determine coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Determine coordinates of shaft vertexes
    (multiple-value-bind (*p3x* *p3y* *p4x* *p4y*)
      (compute-arrow-shaft-points)
      (draw-big-outline) ;Outline arrow
      (when *do-the-stripes*
        ;; Bind distance between stripes
        (let ((*stripe-d* *stripe-distance*))
          (stripe-arrowhead) ;Stripe head
          (stripe-big-arrow-shaft)))))) ;Stripe shaft
```

```

(defun draw-arrow ()
  ;; Calculate coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    (draw-outline) ;Outline arrowhead
    (when *do-the-stripes*
      |   ;; Calculate distance between stripes
      |   (let ((*stripe-d* (compute-stripe-d)))
          (stripe-arrowhead) ;Stripe head
          (stripe-arrow-shaft)))) ;Stripe shaft

  |   ;;; Calculates horizontal distance between stripes.
  |   ;;; Distance is a fraction of the distance between stripes for the
  |   ;;; large arrow. The divisor depends on the level of recursion.
  |   ;;; Distance divides length of top edge evenly when possible to
  |   ;;; maintain continuity between head and shaft of arrow.
  |   (defun compute-stripe-d ()
      |   ;; Distance should be at least 3 pixels so that there is some
      |   ;; white space between lines.
      |   (if (<= *stripe-distance* 3)
          |   3
          |   ;; First find a fraction of *STRIPE-DISTANCE* that depends
          |   ;; on recursion level
          |   (loop for dist = (fixr (/ *stripe-distance*
                                   (selectq *depth*
   (0 2)
   (1 4)
   (2 2)
   (3 1.5)
   (4 1.5)
   (otherwise 2))))
              |   ;; Increment if it doesn't divide *TOP-EDGE* evenly
              |   then (1+ dist)
              |   when (= 0 (\ *top-edge* dist))
              |   ;; Stop when no remainder. Don't return a value
              |   ;; less than 3.
              |   do (return (if (<= dist 3) 3 dist))))))

```

3. Replace *\*stripe-distance\** with *\*stripe-d\** in the functions *stripe-arrowhead* and *draw-arrow-shaft-stripes*.



```

(defun stripe-arrowhead ()
  ;; Find x-coord of top of last stripe to be drawn
  (loop with last-x = (- *p0x* *top-edge*)
        ;; Find starting x-coord for each stripe, decrementing
        ;; by distance between stripes. Stop at last x-coord.
        | for start-x from *p0x* by *stripe-d* above last-x
        ;; Find ending y-coord for each stripe, decrementing by
        ;; distance between stripes.
        | for end-y downfrom *p0y* by *stripe-d*
        ;; Draw a stripe
        do (draw-arrowhead-lines start-x end-y)))

(defun draw-arrow-shaft-stripes
  (left-x top-y right-x bottom-y)
  ;; Find y-coord of starting point of stripe. Don't go
  ;; below the bottom of the triangle.
  | (loop for start-y from top-y by *stripe-d* above bottom-y
        ;; Find x-coord of ending point of the stripe
        | for end-x downfrom right-x by *stripe-d*
        ;; Draw a stripe
        do (draw-arrow-shaft-lines
            left-x start-y end-x bottom-y)))

```

We adjust the stripe density in three steps:

1. Define two new constants for each arrow, **\*d1\*** and **\*d2\***. **\*d1\*** represents the stripe density, or the proportion of the distance between stripes that is black, at the upper right of each arrow. **\*d2\*** represents the density at lower left for each arrow. We estimate **\*d1\*** to be 0.15 and **\*d2\*** to be 0.75.

```

| (defconst *d1* 0.15
|   "Proportion of distance between upper right stripes that is black")

| (defconst *d2* 0.75
|   "Proportion of distance between lower left stripes that is black")

```

2. Define a function, **compute-nlines**, that returns the number of adjacent lines that make up a stripe to be drawn. This function calls another, **compute-dens**, to calculate the proportion of the distance between stripes that is black. This proportion is a

function of the position of the stripe between the upper right and lower left of the arrow. `compute-nlines` multiplies this proportion by `*stripe-d*` to determine the number of lines that make up the stripe. This number must be at least one and less than `*stripe-d*` minus one.

The argument to `compute-nlines` represents the horizontal position of the stripe to be drawn between the upper right and lower left of the arrow. Imagine the top edge of each arrow projected to the left beyond the arrowhead. Imagine each stripe projected to the upper left until it intersects with the extended top edge. The argument to `compute-nlines` is the x-coordinate of this intersection. `*p0x*` is the x-coordinate of this intersection for the top right corner of each arrow, where the stripe density is `*d1*`. `*x2*` is the x-coordinate of this intersection for the lower left stripe in each arrow, where the density is `*d2*`. The x-coordinate for each stripe must be between `*p0x*` and `*x2*`, and the density must be between `*d1*` and `*d2*`.

```
| (defvar *x2* nil
|   "X-coordinate of projection of lower left stripe on top edge")

(defun draw-big-arrow ()
  ;; Determine coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Determine coordinates of shaft vertexes
    (multiple-value-bind (*p3x* *p3y* *p4x* *p4y*)
      (compute-arrow-shaft-points)
      (draw-big-outline) ;Outline arrow
      (when *do-the-stripes*
        ;; Bind distance between stripes and x-coord of
        ;; projection of last stripe onto top edge
        (let ((*stripe-d* *stripe-distance*)
              (*x2* (- *p0x* *top-edge* *top-edge*)))
          (stripe-arrowhead) ;Stripe head
          (stripe-big-arrow-shaft)))))) ;Stripe shaft
```

```

(defun draw-arrow ()
  ;; Calculate coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    (draw-outline) ;Outline arrowhead
    (when *do-the-stripes*
      ;; Calculate distance between stripes and x-coord of
      ;; projection of last stripe onto top edge
      (let ((*stripe-d* (compute-stripe-d)
                (*x2* (- *p0x* *top-edge* *top-edge*)))
            (stripe-arrowhead) ;Stripe head
            (stripe-arrow-shaft)))) ;Stripe shaft

|
|   ;;; Calculates the number of lines that compose each stripe.
|   ;;; Calls COMPUTE-DENS to calculate the proportion of distance
|   ;;; between stripes to be filled, then multiplies by the actual
|   ;;; distance between stripes. Makes sure that there is at least
|   ;;; one line and that there aren't too many lines to leave some
|   ;;; white space.
|   (defun compute-nlines (x)
|     ;; Call COMPUTE-DENS and multiply result by *stripe-d*
|     (let ((n1 (fix (* *stripe-d* (compute-dens x)))))
|       ;; Supply at least one line
|       (cond ((≤ n1 1) 1)
|              ;; But leave some white space between lines
|              ((≥ n1 (- *stripe-d* 1)) (- *stripe-d* 2))
|              (t n1))))

|
|   ;;; Calculates proportion of distance filled in between each stripe.
|   ;;; The argument is the x-coordinate of the projection of the current
|   ;;; stripe onto the line formed by the top edge. Determines where the
|   ;;; projection of the current stripe is on this line in relation to the
|   ;;; distance from first to last stripes in the arrow. Multiplies this
|   ;;; fraction by the difference between densities of first and last
|   ;;; stripes. Finally, adds the density of the first stripe.
|   (defun compute-dens (x)
|     (+ *d1* (* (- *d2* *d1*)
|                (// (- x *p0x*) (float (- *x2* *p0x*)))))

```

3. For each function that draws a stripe, replace the sending of one `:show-lines` message by a loop that might send several.

Determine the number of messages each function should send by calling **compute-nlines**.

```
(defun stripe-arrowhead ()
  ;; Find x-coord of top of last stripe to be drawn
  (loop with last-x = (- *p0x* *top-edge*)
        ;; Find starting x-coord for each stripe, decrementing
        ;; by distance between stripes. Stop at last x-coord.
        for start-x from *p0x* by *stripe-d* above last-x
        ;; Find ending y-coord for each stripe, decrementing by
        ;; distance between stripes.
        for end-y downfrom *p0y* by *stripe-d*
        |   ;; Find number of lines in the stripe
        |   for nlines = (compute-nlines start-x)
        |   ;; Draw the lines that make up the stripe
        |   do (draw-arrowhead-lines nlines start-x end-y last-x)))

|
| (defun draw-arrowhead-lines (nlines start-x end-y last-x)
|   ;; Set up a counter
|   (loop for i from 0 below nlines
|         ;; Find starting x-coord, subtracting counter from first
|         ;; x-coord
|         for first-x = (- start-x i)
|         ;; Make sure we don't go past the end of the arrowhead
|         while (< last-x first-x)
|         ;; Draw a line
|         do (send *dest* 'show-lines
|                 first-x *p0y* *p0x* (- end-y i))))
|
```

```

(defun stripe-big-arrow-shaft-left ()
  ;; Set up a counter for depth. Don't exceed maximum recursion
  ;; level.
  (loop for shaft-depth from 0 below *max-depth*
        ;; Find current top edge and its fractions
        for top-edge = *top-edge* then (// top-edge 2)
        for top-edge-2 = (// top-edge 2)
        for top-edge-4 = (// top-edge 4)
        ;; Find coordinates of apex of triangle
        for apex-x = *p2x* then (- apex-x top-edge-2)
        for apex-y = *p2y* then (- apex-y top-edge-2)
        ;; Find x-coord of bottom right vertex
        for right-x = (+ apex-x top-edge-4)
        ;; Find y-coord of bottom edge of triangle
        for bottom-y = (- apex-y top-edge-4)
  |      ;; Find the x-coord of the projection of the first
  |      ;; stripe onto top edge
  |      for xoff = (- *p0x* *top-edge*) then (- xoff top-edge)
  |      ;; Stripe each triangle
  |      do (draw-big-arrow-shaft-stripes-left
  |          top-edge-4 apex-x apex-y right-x bottom-y xoff)))

(defun draw-big-arrow-shaft-stripes-left
  | (top-edge-4 apex-x apex-y right-x bottom-y xoff)
  | (loop with half-distance = (// *stripe-distance* 2)
        ;; Find x-coord of last stripe in triangle
        with last-x = (- apex-x top-edge-4)
        ;; Find x-coord of top of each stripe, decrementing
        ;; from the apex by HALF the horizontal distance
        ;; between stripes. Stop at last stripe.
        for start-x from apex-x by half-distance above last-x
        ;; Find y-coord of top of stripe
        for start-y downfrom apex-y by half-distance
        ;; Find x-coord of endpoint of stripe
        for end-x downfrom right-x by *stripe-distance*
  |      ;; Find number of lines in the stripe
  |      for nlines = (compute-nlines (- xoff (- right-x end-x)))
  |      ;; Draw a stripe
  |      do (draw-big-arrow-shaft-lines-left
  |          nlines start-x start-y end-x bottom-y last-x)))

```

```

(defun draw-big-arrow-shaft-lines-left
|   (nlines start-x start-y end-x end-y last-x)
|   ;; Set up two counters -- we need to draw two lines at once
|   (loop for i from 0
|         for i2 from 0 by 2
|         ;; Find x-coord of top of first line in stripe
|         for first-x = (- start-x i)
|         ;; Don't exceed number of lines in stripe
|         while (< i2 nlines)
|         ;; Don't go past the end of the triangle
|         while (< last-x first-x)
|         ;; Draw a line
|         do (send *dest* ':show-lines first-x (- start-y i)
|               (- end-x i2) end-y)
|         ;; Draw a second line. The two lines are a refinement
|         ;; to stagger the endpoints of the lines so the diagonal
|         ;; edge looks neat.
|         (send *dest* ':show-lines first-x (- start-y i 1)
|               (- end-x i2 1) end-y)))

(defun stripe-big-arrow-shaft-right ()
|   ;; Set up a counter for depth. Don't exceed maximum recursion
|   ;; level.
|   (loop for shaft-depth from 0 below *max-depth*
|         ;; Find new top edge and its fractions
|         for top-edge = *top-edge* then (// top-edge 2)
|         for top-edge-2 = (// top-edge 2)
|         for top-edge-4 = (// top-edge 4)
|         ;; Find coords of top point of triangle
|         for start-x = (+ *p2x* top-edge-4)
|         for top-y = (- *p2y* *top-edge-4*)
|         then (- top-y top-edge-2 top-edge-4)
|         ;; Find x-coord of projection of first stripe onto
|         ;; top-edge
|         for xoff = (- *p0x* *top-edge*) then (- xoff top-edge)
|         ;; Stripe the triangle
|         do (draw-big-arrow-shaft-stripes-right
|             top-edge-2 top-edge-4 start-x top-y xoff)))

```

```

(defun draw-big-arrow-shaft-stripes-right
|   (top-edge-2 top-edge-4 start-x top-y xoff)
|   (loop with half-distance = (// *stripe-distance* 2)
|         ;; Find y-coord of last stripe in triangle
|         with last-y = (- top-y top-edge-2)
|         ;; Find y-coord of starting point of stripe. Don't go
|         ;; past the end of the triangle.
|         for start-y from top-y by *stripe-distance* above last-y
|         ;; Find coords of ending point of the stripe, decrementing
|         ;; by HALF the horizontal distance between stripes
|         for end-x downfrom (+ start-x top-edge-4) by half-distance
|         for end-y downfrom (- top-y top-edge-4) by half-distance
|         ;; Find number of lines that make up the stripe
|         for nlines = (compute-nlines (- xoff (- top-y start-y)))
|         ;; Draw a stripe
|         do (draw-big-arrow-shaft-lines-right
|             nlines start-x start-y end-x end-y last-y)))

(defun draw-big-arrow-shaft-lines-right
|   (nlines start-x start-y end-x end-y last-y)
|   ;; Set up two counters -- we need to draw two lines at once
|   (loop for i from 0
|         for i2 from 0 by 2
|         ;; Find y-coord of ending point of line
|         for stop-y = (- end-y i)
|         ;; Don't exceed number of lines in the stripe
|         while (< i2 nlines)
|         ;; Don't go past the bottom of the triangle
|         while (< last-y stop-y)
|         ;; Draw a line
|         do (send *dest* ':show-lines start-x (- start-y i2)
|                 (- end-x i) stop-y)
|         ;; Draw a second line. The two lines are a refinement
|         ;; to stagger the endpoints of the lines so the diagonal
|         ;; edge looks neat.
|         (send *dest* ':show-lines start-x (- start-y i2 1)
|               (- end-x i 1) stop-y)))

```

```

(defun stripe-arrow-shaft ()
  ;; Set up a counter for depth. Don't exceed maximum
  ;; recursion level.
  (loop for shaft-depth from *depth* below *max-depth*
        ;; Calculate fractions of new top edge
        for top-edge-2 = *top-edge-2* then (// top-edge-2 2)
        for top-edge-4 = (// top-edge-2 2)
        ;; Find coords of top left point of triangle
        for left-x = *p2x* then (- left-x top-edge-4)
        for top-y = *p2y* then (- top-y top-edge-2 top-edge-4)
        ;; Find coords of bottom right point of triangle
        for right-x = (+ left-x top-edge-2)
        for bottom-y = (- top-y top-edge-2)
  |   ;; Find x-coord of projection of first stripe onto top edge
  |   for xoff = (- *p0x* *top-edge*)
  |   then (- xoff top-edge-2 top-edge-2)
  |   ;; Stripe the triangle
  |   do (draw-arrow-shaft-stripes
  |       left-x top-y right-x bottom-y xoff)))

(defun draw-arrow-shaft-stripes
  |   (left-x top-y right-x bottom-y xoff)
  |   ;; Find y-coord of starting point of stripe. Don't go
  |   ;; below the bottom of the triangle.
  |   (loop for start-y from top-y by *stripe-distance* above bottom-y
        ;; Find x-coord of ending point of the stripe
        for end-x downfrom right-x by *stripe-d*
  |   ;; Find number of lines in the stripe
  |   for nlines = (compute-nlines (- xoff (- right-x end-x)))
  |   ;; Draw a stripe
  |   do (draw-arrow-shaft-lines
  |       nlines left-x start-y end-x bottom-y)))

```



```

| (defun draw-arrow-shaft-lines
|   (nlines left-x start-y end-x bottom-y)
|   ;; Set up a counter. Don't exceed number of lines in the stripe.
|   (loop for i from 0 below nlines
|         ;; Find x-coord of ending point of the line
|         for last-x = (- end-x i)
|         ;; Don't go past the left edge of the triangle
|         while (< left-x last-x)
|         ;; Draw a line
|         do (send *dest* ':show-lines left-x (- start-y i)
|                 last-x bottom-y)))

```

Figure 4 shows the output of the program with stripes of varying spacing and thickness.

At this stage in developing the program we define new functions, constants, and variables. But most of the work consists of changing existing code. Often you need to make similar changes to several functions: you add an argument or replace sending one message by a loop that sends several. In this case we are refining a new program, but when maintaining existing code you must also make selective or global changes. The most helpful facilities are those for finding out about existing code and for editing code. See the section "Finding Out About Existing Code", page 260. See the section "Editing Code", page 282.

## 28.8 Editing Code

Some features are useful mainly in composing new code. See the section "Preparing to Write Code", page 233. See the section "Keeping Track of Lisp Syntax", page 247. Other features are helpful in both writing and editing code. See the section "Finding Out About Existing Code", page 260. In this section we discuss features that are likely to be most useful in editing existing code.

---

### 28.8.1 Identifying Changed Code

Two pairs of List and Edit commands find or edit changed definitions in buffers or files. By default, the commands find changes made since the file was read; use numeric arguments to find definitions that have changed since they were last compiled or saved.

---

#### Example

After defining the routine that calculates the number of lines that compose each stripe, we changed many functions to call that routine and draw the appropriate number of lines. We want to

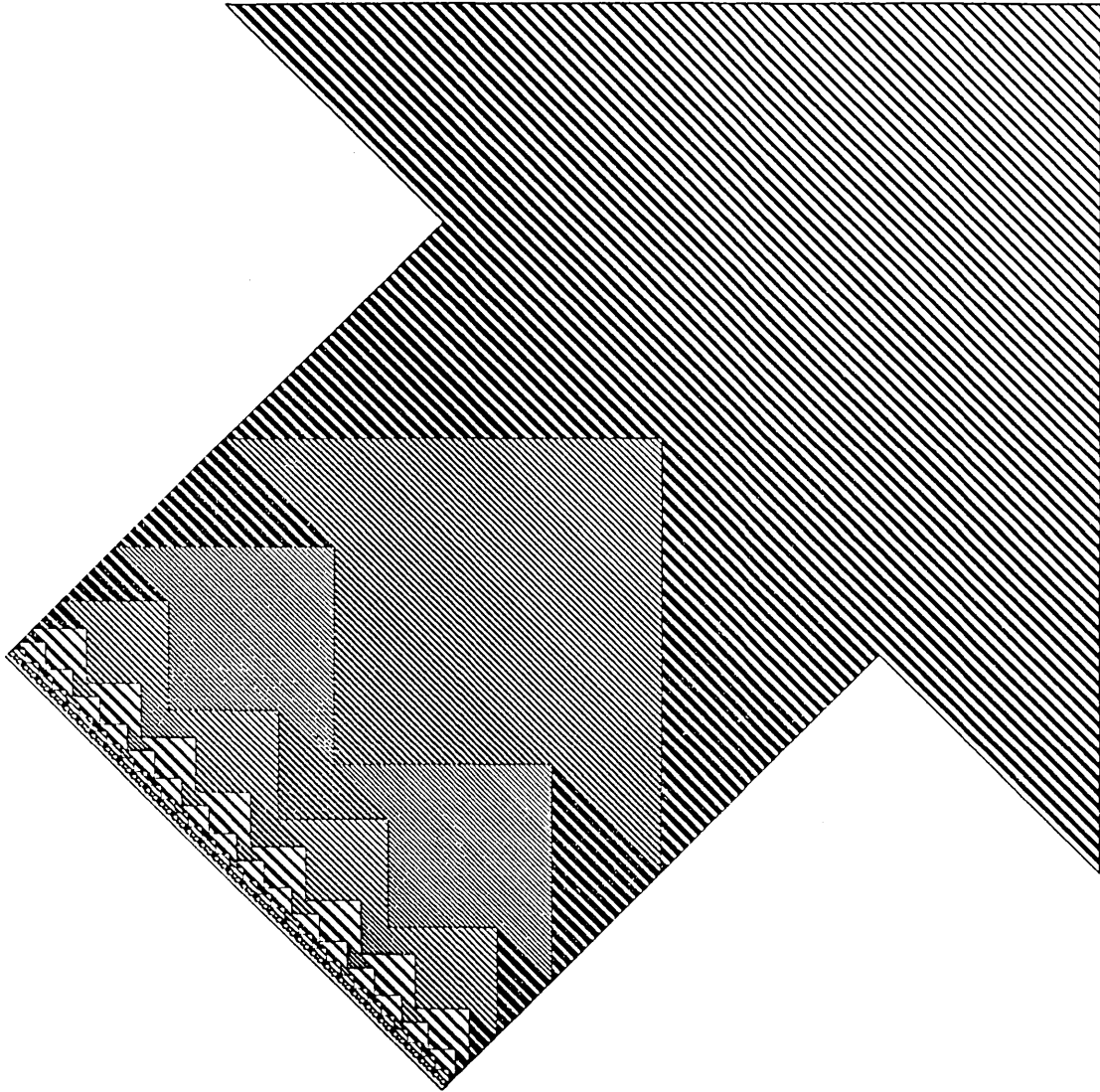


Figure 4. Program output with stripes of varying spacing and density.

look over the changes before recompiling the edited definitions. We use Edit Changed Definitions Of Buffer (m-X).

---

### Reference

|                                          |                                                                                                                        |
|------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| List Changed Definitions Of Buffer (m-X) | Lists definitions in the buffer that have changed since the file was read. Press c-. to edit the definitions listed.   |
| Edit Changed Definitions Of Buffer (m-X) | Prepares for editing definitions in the buffer that have changed. Press c-. to edit subsequent definitions.            |
| List Changed Definitions (m-X)           | Lists definitions in any buffer that have changed since the files were read. Press c-. to edit the definitions listed. |
| Edit Changed Definitions (m-X)           | Prepares for editing definitions in any buffer that have changed. Press c-. to edit subsequent definitions.            |
| Print Modifications (m-X)                | Displays lines in the current buffer that have changed since the file was read.                                        |
| Source Compare (m-X)                     | Compares two buffers or files, listing differences.                                                                    |
| Source Compare Merge (m-X)               | Compares two buffers or files and merges differences into a buffer.                                                    |

---

### 28.8.2 Searching and Replacing

Some facilities discussed elsewhere, particularly the series of List and Edit commands, are useful for displaying and moving to code you wish to edit. See the section "Finding Out About Existing Code", page 260. The commands we discuss here find and replace strings. *Tag tables* offer a convenient means of making global changes to programs stored in more than one file. Use Select All Buffers As Tag Table (m-X) to create a tag table for all buffers

read in. Use Select System As Tag Table (m-X) to create a tag table for all files in a system. For information on systems: See the section "Maintaining Large Programs", page 141.

---

### Example

We have defined **\*stripe-d\***, and we want to replace some occurrences of the constant **\*stripe-distance\*** by the variable **\*stripe-d\***. We use Query Replace (m-Z) to find each occurrence of **\*stripe-distance\***. By pressing SPACE, we replace **\*stripe-distance\*** by **\*stripe-d\*** in functions like **stripe-arrowhead**. By pressing RUBOUT, we leave **\*stripe-distance\*** in place in functions like **draw-big-arrow-shaft-stripes-left**.

---

### Reference

|                           |                                                                                                                                                                                                                                                                                                                                     |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| List Matching Lines (m-X) | Displays the lines (following point) in the current buffer that contain a string.                                                                                                                                                                                                                                                   |
| Incremental Search (c-S)  | Prompts for a string and moves forward to its first occurrence in the buffer. Press c-S to repeat the search with the same string. Press c-R to search backward with the same string. After you invoke the command, if c-S is the first character you type (instead of a string), uses the string specified in the previous search. |
| Reverse Search (c-R)      | Prompts for a string and moves backward to its last occurrence in the buffer. Press c-R to repeat the search with the same string. Press c-S to search forward with the same string. After you invoke the command, if c-R is the first character you type (instead of a string), uses the string specified in the previous search.  |
| Tags Search (m-X)         | Searches for a string in all files listed in a tag table.                                                                                                                                                                                                                                                                           |

|                                       |                                                                                                                                                      |
|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Replace String (c-?)                  | In the buffer, replaces all occurrences (following point) of one string by another.                                                                  |
| Query Replace (m-?)                   | In the buffer, replaces occurrences (following point) of one string by another, querying before each replacement. Press HELP for possible responses. |
| Tags Query Replace (m-X)              | In files listed in a tag table, replaces occurrences of one string by another, querying before each replacement.                                     |
| Select All Buffers As Tag Table (m-X) | Creates a tag table for all buffers in Zmacs.                                                                                                        |
| Select System As Tag Table (m-X)      | Creates a tag table for files in a system defined by defsystem.                                                                                      |

---

### 28.8.3 Moving Text

#### 28.8.3.1 Moving Through Text

To move short distances through text, you can use the Zmacs commands for moving by lines, sentences, paragraphs, Lisp forms, and screens, or you can click left to move point to the mouse cursor. To move longer distances, you can move to the beginning or end of the buffer or use the scroll bar. To go to another buffer, use Select Buffer (c-X B). To switch back and forth between two buffers, use Select Previous Buffer (c-m-L).

Suppose you want to record a location of point so that you can return to that location later. Two techniques are particularly useful:

- Store the location of point in a register. Use Save Position (c-X S) to store point in a register. Use Jump to Saved Position (c-X J) to return to that location.
- Use m-SPACE to push the location of point onto the mark stack. Later, you can use c-m-SPACE to exchange point and the top of the mark stack. c-U c-SPACE pops the mark stack; repeated execution moves to previous marks. Note: Some Zmacs

commands other than `c-SPACE` push point onto the mark stack. When point is pushed onto the mark stack, the notification "Point pushed" appears below the mode line.

---

### Reference

|                                                   |                                                                                                                                                                           |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Select Buffer ( <code>c-X B</code> )              | Moves to another buffer, reading the buffer name from the minibuffer. With a numeric argument, creates a new buffer.                                                      |
| Select Previous Buffer ( <code>c-m-L</code> )     | Moves to the previously selected buffer.                                                                                                                                  |
| Save Position ( <code>c-X S</code> )              | Stores the position of point in a register. Prompts for a register name.                                                                                                  |
| Jump To Saved Position ( <code>c-X J</code> )     | Moves point to a position stored in a register. Prompts for a register name.                                                                                              |
| Set Pop Mark ( <code>c-SPACE</code> )             | With no argument, sets the mark at point and pushes point onto the mark stack. With an argument of <code>c-U</code> , pops the mark stack.                                |
| Push Pop Point Explicit ( <code>m-SPACE</code> )  | With no argument, pushes point onto the mark stack without setting the mark. With an argument <i>n</i> , exchanges point with the <i>n</i> th position on the mark stack. |
| Move To Previous Point ( <code>c-m-SPACE</code> ) | Exchanges point and the top of the mark stack.                                                                                                                            |
| Swap Point And Mark ( <code>c-X c-X</code> )      | Exchanges point and mark. Activates the region between point and mark. Use Beep ( <code>c-G</code> ) to turn off the region.                                              |

### 28.8.3.2 Killing and Yanking

When you need to repeat text, you usually want to copy it rather than type it again. The most common facilities for copying text are the commands for killing and yanking. Commands that kill more than one character of text push the text onto the kill ring. `c-Y` yanks the last kill into the buffer. After a `c-Y` command, `m-Y` deletes the text just inserted, yanks the previous kill, and rotates the kill ring.

---

#### Example

In the function `draw-big-arrow-shaft-lines-left`, we send two `:show-lines` messages on each iteration. The purpose is to arrange the starting points of the lines along the diagonal edge so that they lie as closely as possible on a 45-degree line. The second `send` expression is nearly identical to the first. Instead of typing a new expression, we copy and edit the first one. We follow these steps:

1. Position the cursor after the close parenthesis that ends the first `send` expression.

```
(defun draw-big-arrow-shaft-lines-left
  (nlines start-x start-y end-x end-y last-x)
  .
  .
  do (send *dest* ':show-lines first-x (- start-y i)
          (- end-x i2) end-y)
```

2. Use `c-m-RUBOUT` to kill the `send` expression and push it onto the kill ring.

```
(defun draw-big-arrow-shaft-lines-left
  (nlines start-x start-y end-x end-y last-x)
  .
  .
  do
```

3. Use `c-Y` to restore the expression.

```
(defun draw-big-arrow-shaft-lines-left
  (nlines start-x start-y end-x end-y last-x)
  .
  .
  do (send *dest* ':show-lines first-x (- start-y i)
          (- end-x i2) end-y)
```

4. Use `LINE` to move to the next line and indent.
5. Use `c-Y` to insert a copy of the `send` expression.

```
(defun draw-big-arrow-shaft-lines-left
  (nlines start-x start-y end-x end-y last-x)
  .
  .
  do (send *dest* ':show-lines first-x (- start-y i)
          (- end-x i2) end-y)
  (send *dest* ':show-lines first-x (- start-y i)
          (- end-x i2) end-y)
```

6. Edit the second `send` expression.

```
(defun draw-big-arrow-shaft-lines-left
  (nlines start-x start-y end-x end-y last-x)
  .
  .
  do (send *dest* ':show-lines first-x (- start-y i)
          (- end-x i2) end-y)
  (send *dest* ':show-lines first-x (- start-y i 1)
          (- end-x i2 1) end-y)))
```

---

### Example

Suppose we have an existing program in which we have already defined the function `compute-nlines`. We can copy the function in three ways:

- Use `c-m-K` or `c-m-RUBOUT` to kill the definition. Use `c-Y` to restore it. Go to the new buffer. Use `c-Y` to insert a copy of the definition.



- Use `c-m-H` to mark the definition. Use `m-W` to push it onto the kill ring. Go to the new buffer. Use `c-Y` to insert a copy of the definition.
- Click middle on the first or last parenthesis of the definition to mark the definition. Click `sh-middle` to push it onto the kill ring. Move to the new buffer. Click `sh-middle` to insert a copy of the definition.

---

### Reference

|                                                |                                                                                                                                                                                                                                                                                                                                                        |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Kill Sexp ( <code>c-m-K</code> )               | Kills forward one or more Lisp expressions.                                                                                                                                                                                                                                                                                                            |
| Backward Kill Sexp ( <code>c-m-RUBOUT</code> ) | Kills backward one or more Lisp expressions.                                                                                                                                                                                                                                                                                                           |
| Mark Definition ( <code>c-m-H</code> )         | Puts point and mark around the current definition.                                                                                                                                                                                                                                                                                                     |
| Save Region ( <code>m-W</code> )               | Pushes the text of the region onto the kill ring without killing the text.                                                                                                                                                                                                                                                                             |
| Yank ( <code>c-Y</code> )                      | Pops the last killed text from the kill ring, inserting the text into the buffer at point. With an argument <i>n</i> , yanks the <i>n</i> th entry in the kill ring. Does not rotate the kill ring.                                                                                                                                                    |
| Yank Pop ( <code>m-Y</code> )                  | After a <code>c-Y</code> command, deletes the text just inserted, yanks previously killed text from the kill ring, and rotates the kill ring. Repeated execution yanks previous kills and rotates the kill ring.                                                                                                                                       |
| [ <i>Region</i> (M2)]                          | When <i>region</i> is defined, pushes the text of <i>region</i> onto the kill ring without killing the text (like <code>m-W</code> ). Repeated execution has the following effects: <ul style="list-style-type: none"> <li>• First repetition: kills the text of <i>region</i>, pushing the text onto the kill ring (like <code>c-W</code>)</li> </ul> |

- Second repetition: pops the text of *region* from the kill ring, inserting the text into the buffer at point (like `c-Y`)
- Third and subsequent repetitions: delete the text just inserted, yank previously killed text from the kill ring, and rotate the kill ring (like `m-Y`)

If no region is defined, pops the last killed text from the kill ring, inserting the text into the buffer at point (like `c-Y`).

Repeated execution deletes the text just inserted, yanks previously killed text from the kill ring, and rotates the kill ring (like `m-Y`).

### 28.8.3.3 Using Registers

Using `c-Y` and `m-Y` to copy text can become tedious when you have to rotate through a long kill ring to find the text you need. Another method, especially useful when you want to copy a piece of text more than once, is to save and restore the text using registers.

---

#### Reference

|                                          |                                                                                               |
|------------------------------------------|-----------------------------------------------------------------------------------------------|
| Put Register ( <code>c-X X</code> )      | Copies contents of the region to a register. Prompts for a register name.                     |
| Open Get Register ( <code>c-X G</code> ) | Inserts contents of a register into the current buffer at point. Prompts for a register name. |

### 28.8.3.4 Copying Buffers and Files

Use Insert File (`m-X`) to place the contents of an entire file in your buffer.

You can copy the contents of a buffer in two ways:

- Use Insert Buffer (`m-X`), naming the buffer you want to copy.

- Use `c-X H` to mark the buffer you want to copy. Use `m-W` to push its text onto the kill ring. Move to the new buffer. Use `c-Y` to insert a copy of the text.

---

### Reference

|                                    |                                                                            |
|------------------------------------|----------------------------------------------------------------------------|
| Mark Whole ( <code>c-X H</code> )  | Marks an entire buffer.                                                    |
| Insert Buffer ( <code>m-X</code> ) | Inserts contents of the specified buffer into the current buffer at point. |
| Insert File ( <code>m-X</code> )   | Inserts contents of the specified file into the current buffer at point.   |

---

## 28.8.4 Keyboard Macros

Sometimes you need to perform a uniform sequence of commands on several pieces of text. You can save keystrokes by converting the sequence to a keyboard macro and installing it on a single key. If you anticipate using a macro often, you can write Lisp code to define it in an init file. If you frequently use particular extended commands, install them on single keys with Set Key (`m-X`).

---

### Reference

|                                            |                                                                                                                               |
|--------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Start Kbd Macro ( <code>c-X (</code> )     | Begins recording keystrokes as a keyboard macro.                                                                              |
| End Kbd Macro ( <code>c-X )</code> )       | Stops recording keystrokes as a keyboard macro.                                                                               |
| Call Last Kbd Macro ( <code>c-X E</code> ) | Executes the last keyboard macro.                                                                                             |
| Name Last Kbd Macro ( <code>m-X</code> )   | Gives the last keyboard macro a name.                                                                                         |
| Install Macro ( <code>m-X</code> )         | Installs on a key the last keyboard macro or a named macro.                                                                   |
| Install Mouse Macro ( <code>m-X</code> )   | Installs a keyboard macro on a mouse click (such as L2). When you click to call the macro, point moves to the position of the |

|                       |                                                                                       |
|-----------------------|---------------------------------------------------------------------------------------|
|                       | mouse cursor before the macro is executed.                                            |
| Deinstall Macro (m-X) | Deinstalls a keyboard macro from a key or a mouse click.                              |
| Set Key (m-X)         | Installs an extended command on a single key. Use HELP C to look for unassigned keys. |

---

## 28.8.5 Using Multiple Windows

### 28.8.5.1 Multiple Buffers

Sometimes when editing you move often between two buffers. You might want to see the two buffers at the same time rather than switch between them. A common use of multiple-window display is to edit source code while viewing compiler warnings. See the section "Using the Compiler Warnings Database", page 309.

---

#### Example

We add a new **:show-lines** message to the program but forget what arguments the message takes. We want to display the source code for the message handler on the same screen as our program code. We use c-X 2 to create another window and move to it. We use Edit Methods (m-X) to find the source code for the method that handles **:show-lines**. See the section "Methods", page 378.

---

#### Example

After finishing the program, we collect a file of bug reports from users. We want to use these reports in correcting our program code. We create two windows, one displaying the program code and the other the bug-report file. We edit the program code, using c-m-V to scroll the bug-report window as we correct each bug.

---

#### Reference

|                     |                                                                                    |
|---------------------|------------------------------------------------------------------------------------|
| Split Screen (m-X)  | Pops up a menu of buffers and splits the screen to display the buffers you select. |
| Two Windows (c-X 2) | Creates a second window, with the current buffer on top and                        |

|                              |                                                                                                                                   |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
|                              | the previous buffer on the bottom. Puts the cursor in the bottom window.                                                          |
| View Two Windows (c-X 3)     | Creates a second window, with the current buffer on top and the previous buffer on the bottom. Puts the cursor in the top window. |
| Modified Two Windows (c-X 4) | Creates a second window and visits a buffer, file, or tag there. Displays the current buffer in the top window.                   |
| Other Window (c-X 0)         | Moves to the other of two windows.                                                                                                |
| Scroll Other Window (c-m-U)  | Scrolls the other of two windows.                                                                                                 |
| One Window (c-X 1)           | Returns to one-window display, selecting the buffer the cursor is in.                                                             |

### 28.8.5.2 Displaying Zmacs and Other Windows

Use [Split Screen] or [Edit Screen] from the System menu to display an editor window on the screen with other kinds of windows.

---

#### Example

In testing new program functions, we want to have the current version of the figure on the same screen as the program code. We use [Split Screen] from the System menu to add a Lisp Listener to the screen. We move between windows by clicking left on the window to which we want to move.

We evaluate (pkg-goto 'graphics) and then (do-arrow) in the Lisp Listener. We adjust the arguments to **draw-arrow-graphic** so that the arrow fits neatly into the Lisp Listener window.

```
(defun do-arrow ()
  (let ((*dest* (make-instance 'screen-arrow-output)))
    (send terminal-io ':clear-screen)
    (draw-arrow-graphic 640 1300 1850)))
```

Figure 5 shows the appearance of the screen with graphic output in a Lisp Listener and source code in a Zmacs buffer.

To return to displaying only the Zmacs window, we use [Split Screen] with the existing Zmacs buffer as the only element.

---

**Reference**

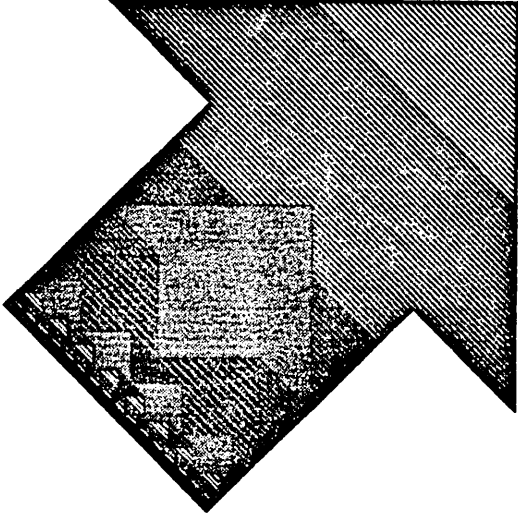
[Split Screen / Lisp / Existing Window / *Existing Zmacs Buffer* / Do It] (from the System menu)  
Adds a Lisp Listener to a screen displaying an existing Zmacs buffer.

[Split Screen / Existing Window / *Existing Zmacs Buffer* / Do It] (from the System menu)  
Resumes one-window display of an existing Zmacs buffer.

**28.8.5.3 Other Displays**

The window system allows you to use menus, choose-variable-values windows, and other multiple-window displays in executing programs. For details: See the section "Using the Window System" in *Programming the User Interface, Volume B*. See the section "Window System Choice Facilities" in *Programming the User Interface, Volume B*. For examples of simple uses of windows, including choose-variable-values windows: See the section "Using Flavors and Windows", page 343.

```

NIL
■

Lisp Listener 2
;;; Calculates the number of lines that compose each stripe.
;;; Calls COMPUTE-DENS to calculate the proportion of distance
;;; between stripes to be filled, then multiplies by the actual
;;; distance between stripes. Makes sure that there is at least
;;; one line and that there aren't too many lines to leave some
;;; white space.
(defun compute-nlines (x)
  ;; Call COMPUTE-DENS and multiply result by *STRIPE-D*
  (let ((n1 (fix (* *stripe-d* (compute-dens x)))))
    ;; Supply at least one line
    (cond ((< n1 1) 1)
          ;; But leave some white space between lines
          ((> n1 (- *stripe-d* 1)) (- *stripe-d* 2))
          (t n1))))

;;; Calculates proportion of distance filled in between each stripe.
;;; The argument is the x-coordinate of the projection of the current
;;; stripe onto the line formed by the top edge. Determines where the
;;; projection of the current stripe is on this line in relation to the
;;; distance from first to last stripes in the arrow. Multiplies this
;;; fraction by the difference between densities of first and last
;;; stripes. Finally, adds the density of the first stripe.
(defun compute-dens (x)
  (+ *d1* (* (- *d2* *d1*)
             (/ (- x *p0x*) (float (- *x2* *p0x*))))))

ZMACS (LISP) pcodex.1 /doss/doc/workstyles/ VIXEN: * [More above and below]

```

L:Move point, L2:Move to point, H:Mark thing, M2:Save/Kill/Yank, R:Menu, R2:System menu.  
 08/17/83 18:06:25 rom GRAPHICS: Ty1\_\_

Figure 5. Using multiple windows to test the program while viewing the source code.

## 29. Compiling and Evaluating Lisp

When should you compile code, and when evaluate it?

The main job of the compiler is to convert interpreted functions into compiled functions. An interpreted function is a list whose first element is **lambda**, **zl:named-lambda**, **zl:subst**, or **zl:named-subst**. These functions are executed by the Lisp evaluator. The most common interpreted functions you define are **zl:named-lambdas**. When you load a source file that contains **defun** forms or when you otherwise evaluate these forms, you create **zl:named-lambda** functions and define the function specs named in the forms to be those functions.

Compiled functions are Lisp objects that contain programs in the instruction set (the machine language). They are executed directly by the microcode. Compiling an interpreted function (by calling the compiler on a function spec) converts it into a compiled function and changes the definition of the function spec to be that compiled function.

You seldom compile functions directly. Instead, you compile either regions of Zmacs buffers or source files.

- Compiling a region of a Zmacs buffer (or the whole buffer) causes the compiler to process the forms in the region, one by one. This processing has side effects on the Lisp environment. For a summary of the compiler's actions: See the section "Compiling Code in a Zmacs Buffer", page 299.
- Compiling a source file translates it into a binary file. With some exceptions, this processing does not have side effects on the Lisp environment at compile time. When you load a compiled file that defines functions, you create compiled rather than interpreted functions and define function specs to be those compiled functions. In other respects, loading a compiled file has essentially the same effects as loading a source file (evaluating the forms in the file). For a discussion of compiling files: See the section "Compiling and Loading a File", page 301.

Most Symbolics programmers compile all their program code. The compiler checks extensively for errors and issues warnings that help detect bugs like typographical errors, unbound symbols, and faulty Lisp syntax. Compiled code runs faster and takes up less storage than interpreted code. You can compile code in portions and decide at compile time whether or not to save the compiler output in a binary file.

The most common use for interpreted functions is stepping through their execution. You cannot step through the execution of a compiled function. If a function is compiled, you can read its definition into a Zmacs buffer, evaluate the definition, and then step through a function call.

In addition to evaluating definitions to create interpreted functions, you might need to evaluate forms to test a program or find information about a Lisp object.



(Unless you are using the Stepper, functions that you call during these evaluations are usually compiled.) You can evaluate a form in a Lisp Listener, a breakpoint loop, or the minibuffer.

For more information on functions: See the section "Functions" in *Symbolics Common Lisp: Language Concepts*.

## 29.1 Compiling Lisp Code

You can use Zmacs commands to compile code in a file or Zmacs buffer. Most Symbolics programmers compile code as soon as they have written enough to test. This practice lets them correct errors quickly and produce simple working versions of programs before adding more complex operations. A common command for incremental compiling from a Zmacs buffer is Compile Region (`c-sh-C`). If no region is defined, this command compiles the current definition.

In addition to compiling definitions as they write them, Symbolics programmers consider it good practice to recompile a function soon after effecting a change. Because recompiling a series of functions or an entire program can be time-consuming, it is easier and faster to make changes and then use a single command to recompile only the changed functions. Using Compile Changed Definitions Of Buffer (`m-sh-C`) or Compile Changed Definitions (`m-X`) is easier in this case than recompiling each function separately or recompiling the entire buffer.

The order in which you compile definitions can be important. For example, suppose you have a function that binds a variable you want to be treated as special. If you compile the function definition before compiling the variable declaration, the compiler treats the variable as local and generates incorrect output. For this reason you should usually put `defvar` and `zl:defconst` forms at the beginning of a file or into a separate file to be compiled and loaded before function definitions.

When editing a program, it is a good idea to load the entire program before you start work on it. When you compile new definitions or recompile edited ones, the compiler will have access to variable declarations, macros, functions, and other information. You will also be able to use Zmacs commands and Lisp functions for finding information about other parts of the program. See the section "Finding Out About Existing Code", page 260.

Sometimes when you compile a file, write large sections of code at once, or make many changes to a large program, compiling the code produces many warning messages. For a description of how Edit Compiler Warnings (`m-X`) lets you use the compiler warnings as a reference source for debugging: See the section "Debugging Lisp Programs", page 309.

For more information on the compiler: See the section "The Compiler", page 105.

---

### 29.1.1 Compiling Code In a Zmacs Buffer

Compiling a top-level form in a Zmacs buffer – using a command like `Compile Region (c-sh-C)` or `Compile Buffer (m-X)` – has side effects on the Lisp environment. Following is a summary of the compiler's actions:

| <i>Form</i>                        | <i>Action</i>                                                                                                                                                                                                                                                                                                                                |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Macro form                         | If the form is a list whose first element is a macro, the compiler expands the form and processes this expanded form instead of the original.                                                                                                                                                                                                |
| Function definition                | If the form is a list whose first element is <code>defun</code> , the compiler constructs a lambda-expression from the definition, converts the lambda-expression into a compiled function, and defines the function spec named in the definition to be that compiled function.                                                              |
| Macro definition                   | If the form is a list whose first element is <code>macro</code> , the compiler constructs a lambda-expression as the macro's expander function, converts the lambda-expression into a compiled function, and defines the function spec named in the definition to be the macro. A <code>defmacro</code> form expands into this kind of form. |
| Special case                       | Some forms, like <code>eval-when</code> , <code>declare</code> , and <code>progn 'compile</code> forms, have special meaning for the compiler. It handles each of these in a different way. For details: See the section "How the Stream Compiler Handles Top-level Forms", page 111.                                                        |
| Atom, <code>zl:comment</code> form | The form is ignored.                                                                                                                                                                                                                                                                                                                         |
| Other                              | The form is evaluated.                                                                                                                                                                                                                                                                                                                       |

---

### Example

We have written some initial code for the controlling function of the calculation module:

```
(defvar *top-edge* nil
  "Length of the top edge of the arrow")

(defvar *p0x* nil
  "X-coordinate of point 0")

(defvar *p0y* nil
  "Y-coordinate of point 0")

(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4)))
    (draw-big-arrow)))
```

Because we have no other code in the buffer, we can compile these definitions using Compile Buffer (*m-x*). If we had more code in the buffer, we could compile these definitions by setting the mark at one end and point at the other and using Compile Region (*c-sh-C*).

The compiler displays the following warnings:

```
For Function DRAW-ARROW-GRAPHIC
  The variable *TOP-EDGE-4* was never used.
  The variable *TOP-EDGE-2* was never used.
  The variable *P0X* was never used.
```

```
The following functions were referenced but don't seem defined:
DRAW-BIG-ARROW referenced by DRAW-ARROW-GRAPHIC
```

The first set of warnings indicates that the compiler is treating **\*top-edge-2\***, **\*top-edge-4\***, and **\*p0x** as local variables. We neglected to declare **\*top-edge-2\*** and **\*top-edge-4\*** special with **defvar**; **\*p0x** is of course a misspelling. The lack of a definition for **draw-big-arrow** is not surprising; we have yet to write that definition.

We add the two **defvars** and correct the spelling of **\*p0x\***. We compile the changes using Compile Changed Definitions Of Buffer (*m-sh-C*). The compiler now displays only one warning:

The following functions were referenced but don't seem defined:  
 DRAW-BIG-ARROW referenced by DRAW-ARROW-GRAPHIC

We continue writing the program by defining **draw-big-arrow**.

---

### Reference

Compile Region (c-sh-C)                      Compiles the region. If no region is marked, compiles the current definition.

[Zmacs Window / Compile Region]

Compiles the region. If no region is marked, compiles the current definition.

Compile Changed Definitions Of Buffer (m-sh-C)

Compiles all the definitions in the current Zmacs buffer that have changed since the definitions were last compiled.

Compile Changed Definitions (m-X)

Compiles all the definitions in any Zmacs buffer that have changed since the definitions were last compiled.

Compile Buffer (m-X)

Compiles the current Zmacs buffer.

Compile (m-X) [Zmacs Window (R)]

Pops up a menu of options for compiling code in the current context.

---

### 29.1.2 Compiling and Loading a File

Compiling a file, using Compile File (m-X) or `compiler:compile-file`, saves the compiler output in a binary file of canonical type `:bin`. For the most part, compiling a file does not have side effects on the Lisp environment. The basic difference between compiling a source file and compiling the same forms in a buffer is this: When you compile a file, most function specs are not defined and most forms (except those within

**eval-when (compile)** forms) are not evaluated at compile time. Instead, the compiler puts instructions into the binary file that cause these things to happen at load time. You can load a source or binary file into the Lisp environment by using Load File (m-X) or **zl:load**. You can compile a file and then load the resulting binary file by using **compiler:compile-file-load**.

---

### Example

In a previous session, we wrote the output routines for the program, saved them in a file, and compiled that file. Now we are writing the first calculation routines, and we want to test them. We need to load the file that contains the compiled code for the output routines. We use Load File (m-X).

Suppose our two files are in the directory >Symbolics>examples> on Lisp Machine acme-blue. The file containing the output routines is arrow-out. The current Zmacs buffer, and the file containing the calculation module, is arrow-calc. When we type m-X load file (or m-X lo f, using completion), Zmacs prompts for a file name:

Load File: (Default is ACME-BLUE:>Symbolics>examples>arrow-calc)

We type arrow-out, without a file type. The latest version of arrow-out.bin is loaded. If no compiled version exists or if the latest compiled file is older than the latest source file, Zmacs offers to compile the source file and then load the compiled version.

---

### Reference

- |                                          |                                                                                                                                                                |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Compile File (m-X)                       | Prompts for the name of a file and compiles that file, placing the compiled code in a file of canonical type <b>:bin</b> .                                     |
| <b>(compiler:compile-file file-name)</b> | Compiles a file, placing the compiled code in a file of canonical type <b>:bin</b> .                                                                           |
| Load File (m-X)                          | Prompts for a file name, taking the default from the current buffer. Offers to save the buffer if it has changed since the file was last read or saved. Offers |

to compile the source file if no compiled version exists or if the source file was created after the latest compiled version. If you specify a file type, loads the latest version of the file of that type. If you don't specify a file type, loads the latest version of the binary file (even if older than the latest source file); if no binary file exists, loads the latest source file.

**(load *file-name*)**

Loads a file into the Lisp environment. If you specify a file type, loads the latest version of the file of that type. If you don't specify a file type, loads the latest version of the binary file (even if older than the latest source file); if no binary file exists, loads the latest source file.

**(compiler:compile-file-load *file-name*)**

Compiles a file, placing the compiled code in a file of canonical type **:bin**. Loads the resulting binary file.

## 29.2 Evaluating Lisp Code

---

### 29.2.1 Evaluation and the Editor

The most common reason for evaluating definitions in a Zmacs buffer is to step through the execution of the functions they define. Sometimes in debugging you want to proceed step by step through a function call, using **zl:step** or the **:step** option for **trace**. See the section "Tracing and Stepping", page 323. You can do this only with interpreted functions. If a function is compiled, you can use Edit Definition (**m-.)** to read its definition into a Zmacs buffer. You can then evaluate the definition using Evaluate Region (**c-sh-E**). When you have finished stepping, you can recompile the definition.

The evaluation of Lisp forms in the editing buffer or the minibuffer normally displays the returned values in the echo area (beneath the mode line near the bottom of the screen). Any output to **zl:standard-output** during the evaluation appears in the editor typeout window. Two commands, Evaluate Into Buffer (**m-X**) and Evaluate And Replace Into Buffer (**m-X**), print the returned values in the Zmacs buffer at point. With a numeric argument, these commands also insert any typeout from the evaluation into the Zmacs buffer.

Often while editing you need to evaluate forms other than definitions in a buffer. You need to call a function to test your program, or you need to call a function like **describe** to find information about a Lisp object. (Of course, these functions need not be interpreted.) You can type forms to be evaluated in three ways:

- Use **m-ESCAPE** to evaluate a form in the minibuffer.
- Use **SUSPEND** to enter a Lisp breakpoint loop. You type forms that are read in the buffer's package and evaluated. Use **RESUME** to return to the editor.
- Use **SELECT L** or **[Lisp]** from the System menu to select a Lisp Listener and evaluate forms there. Use **SELECT E** or **[Edit]** from the System menu to return to the editor.

---

### Example

We have found a bug in the program and suspect that it lies in the function **do-arrows**. We want to step through a call to that function, but it is compiled. We use Edit Definition (**m-.**) to find the definition of **do-arrows** and Evaluate Region (**c-sh-E**) to evaluate the definition. We then step through a function call. See the section "Stepping", page 325.

---

### Example

We have written and compiled the output routines and the initial code for the calculation module. We want to test the program as written so far. The top-level function to call is **do-arrow**. We can test the program in three ways:

- Press **m-ESCAPE** and evaluate (**do-arrow**). The graphic output appears in a typeout window. We press **SPACE** to restore the editing buffer to the screen.
- Press **SUSPEND** to enter a Lisp breakpoint loop and evaluate (**do-arrow**) there. We press **RESUME** to return to the editor.

- Press SELECT L to select a Lisp Listener. If the current package is not **graphics**, we first evaluate (pkg-goto 'graphics) and then (do-arrow). We press SELECT E to return to the editor.

---

### Example

We want to be sure that new function names do not conflict with other symbol names in the **graphics** package. Most of our function names contain the string "arrow". We want to find the symbol names that contain that string. We use m-ESCAPE, SUSPEND, or SELECT L and evaluate:

(apropos "arrow" 'graphics)

---

### Reference

|                                                 |                                                                                                                        |
|-------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| Evaluate Region (c-sh-E)                        | Evaluates the region. If no region is marked, evaluates the current definition.                                        |
| Evaluate Changed Definitions Of Buffer (m-sh-E) | Evaluates all the definitions in the current Zmacs buffer that have changed since the definitions were last evaluated. |
| Evaluate Changed Definitions (m-X)              | Evaluates all the definitions in any Zmacs buffer that have changed since the definitions were last evaluated.         |
| Evaluate Buffer (m-X)                           | Evaluates the current Zmacs buffer.                                                                                    |
| Evaluate Into Buffer (m-X)                      | Prompts for a Lisp form to evaluate and prints the returned values in the Zmacs buffer at point.                       |
| Evaluate And Replace Into Buffer (m-X)          | Evaluates the Lisp form following point and replaces it with the printed representation of the values it returns.      |



|                                           |                                                                                                                                                                                               |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Evaluate Minibuffer (M-ESCAPE)            | Prompts for a Lisp form to evaluate in the minibuffer and displays the returned values in the echo area.                                                                                      |
| Evaluate (M-X) [ <i>Zmacs Window</i> (R)] | Pops up a menu of options for evaluating code in the current context.                                                                                                                         |
| SUSPEND                                   | Enters a Lisp breakpoint loop, where you can evaluate forms. The current package in the breakpoint loop is the same as in the previous context. Use RESUME to return to the previous context. |

---

### 29.2.2 Lisp Input Editing

When typing to a Lisp Listener you can use many editing commands to modify a form before you evaluate it. You often repeat the same function calls or variations of similar function calls when testing code. Instead of retyping these forms, you can use the Lisp input editor's ring of input entries to retrieve them within the same Lisp Listener. When you yank a previous form, the Lisp input editor places the cursor at the end of the form but omits the final close parenthesis or carriage return. You can then edit the form before typing the final delimiter to evaluate it.

---

#### Example

We execute our program by calling the function `do-arrow`. We evaluate `(do-arrow)` once and would like to evaluate it again within the same Lisp Listener. We press `C-M-Y` to yank the last form we typed. If that is not `(do-arrow)`, we press `M-Y` until `(do-arrow` appears, without the close parenthesis. We type a close parenthesis to begin the evaluation.

---

#### Reference

`C-M-Y`

Yanks the last form typed to the Lisp Listener. It waits after the final delimiter for you to press `END`, allowing you to edit the form before evaluating it. With

m-Y

an argument *n*, yanks the *n*th form in the input ring. In Zmacs, this command performs a different action: it repeats the last minibuffer command typed.

After a c-m-Y command, deletes the form just inserted, yanks the previous form from the input ring, and rotates the input ring. Repeated execution yanks previous forms and rotates the input ring. In Zmacs, this command rotates either the minibuffer command history or the text kill history, (depending on which yanking command it follows) and yanks elements from that history. See the section "Retrieving History Elements" in *Text Editing and Processing*.



## 30. Debugging Lisp Programs

The General software environment offers you a powerful interactive Debugger and a variety of other tools for debugging Lisp programs. The kind of debugging tool you use depends on the application of the program. Bugs might be more obvious in a graphics programs than in a minor modification of some internal system function. Problems with a graphics programs are sometimes evident from the program's output. On the other hand, programs with a complex window system application might have bugs that are difficult to identify.

Debugging tools are more appropriate for some kinds of bugs than for others. You commonly encounter three sorts of problems with a program:

- The program does not compile correctly. You can use the compiler warnings database to edit code before recompiling.
- The program compiles, but running it signals an error. Usually errors invoke the Debugger, where you can examine stack frames, return values, disassemble code, call the editor, and perform other tasks.
- The program runs but does not behave as it should. You can use many techniques for finding the problem, including commenting out sections of code, tracing, stepping, setting breakpoints, disassembling, and inspecting. Often the most effective method is simply studying the source code.

For complete information on the Debugger: See the section "Debugger", page 3. Also: See the section "Miscellaneous Debugging Aids", page 91.

### 30.1 Using the Compiler Warnings Database

The compiler sometimes produces many warning messages. The compiler maintains a database of these messages, organized by file. Each time you compile or recompile code, the compiler adds or removes warnings from the database, so that the database reflects the state of your program as of the last time you compiled it.

If you want to save warnings in a file, you can use `Compiler Warnings (m-X)` to put them in a buffer and then write them to a file. When you make a system using `make-system`, you can use the `:batch` option to save compiler warnings in a file: See the section "Make-system Keywords". Use `Load Compiler Warnings (m-X)` to load compiler warnings into the database from a file.

If compiler warnings exist in the database, `Edit Compiler Warnings (m-X)` lets you edit source code while consulting the corresponding warnings. The command splits the screen, with compiler warnings in one window and the source code to which the warnings apply in the other. As you finish editing each section of code, you press `c-.`. This displays the next warning in one window and the source code to which the next warning applies in the other window. When you reach the last compiler warning, pressing `c-.` returns the screen to its previous configuration.

---

### Example

Elsewhere we discuss compiling the initial code for the calculation module of the sample program: See the section "Compiling Code in a Zmacs Buffer", page 299. Figure 6 shows the result of using Edit Compiler Warnings (m-X) after compiling the buffer with the initial code. The compiler warnings are in the upper window and the source code in the lower window.

---

### Reference

|                              |                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Edit Compiler Warnings (m-X) | Prepares to edit all source code that has produced compiler warnings. Lists each file whose code produced warnings and asks whether you want to edit that file. Splits the screen, with compiler warnings in the upper window and source code that produced those warnings in the lower window. Use c-. to display subsequent warnings and edit the applicable code. |
| Compiler Warnings (m-X)      | Puts compiler warning messages into a buffer and selects that buffer.                                                                                                                                                                                                                                                                                                |
| Load Compiler Warnings (m-X) | Loads a file containing compiler warning messages into the compiler warnings database.                                                                                                                                                                                                                                                                               |

## 30.2 Using the Debugger

Some errors during execution automatically invoke the Genera Debugger. You can also enter the Debugger explicitly by pressing m-SUSPEND or c-m-SUSPEND. You can also enter the Debugger from within a program by inserting a call to **break** or **zl:dbg** with no arguments into the code and recompiling. You can force a process into the Debugger by calling **zl:dbg** with an argument of *process*. See the section "Using Breakpoints", page 329.

The Debugger is useful for examining stack frames. With Debugger commands, you can see the arguments for the current stack frame, disassemble its code, return a value from it, go up and down the stack, and invoke the editor to edit function definitions. A common Debugger sequence is to disassemble code for the current frame, call the editor to edit and recompile the function, and test the changed function.

```

Warnings for file VIXEN: /doss/doc/workstyles/pcodex.2
■ For Function DRAW-ARROW-GRAPHIC
  The variable *TOP-EDGE-4* was never used.
  The variable *TOP-EDGE-2* was never used.
  The variable *P0X was never used.
  DRAW-BIG-ARROW was referenced but not defined.

*Compiler-Warnings-1*
(defun draw-arrow-graphic (*top-edge* *p0x *p0y*)
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4)))
    (draw-big-arrow)))

pcodex.l /doss/doc/workstyles/ VIXEN:
ZMACS (LISP) pcodex.l /doss/doc/workstyles/ VIXEN: *
Control-. is now Edit warnings for next function.
1 more definition as well
Point pushed

L:Move point, L2:Move to point, M:Mark thing, M2:Save/Kill/Yank, R:Menu, R2:System menu.
08/20/83 16:49:52 rom          GRAPHICS:      TyI

```

Figure 6. Edit Compiler Warnings (m-x) splits the screen. The upper window contains compiler warnings. The lower window contains the source code.

A window-oriented version of the Debugger is the Window Debugger. Invoke it from within the Debugger by entering the `:Window Debugger` command or by pressing `c-m-w`.

We use the variable `*x2*` in computing the thickness of each stripe. `*x2*` is the x-coordinate of the projection of the last stripe in each arrow onto the top edge. We must bind it for each arrow to the difference between the value of `*p0x*` and twice the value of `*top-edge*`.

Suppose that we forget to bind `*x2*` for the big arrow in the function `draw-big-arrow`. The initial value of `*x2*` is `nil`. In the function `compute-dens`, we subtract `*p0x*` from `*x2*`. Because the value of `*x2*` is not a number, we generate an error when we first call the function. The error invokes the Debugger with the name of the function in which the error occurred, the value of the function's arguments, and the following error message:

```
>Trap: The first argument given to SYS:--INTERNAL, NIL, was not a number.
```

The Debugger also displays a listing of *proceed types*, *special commands*, and *restart handlers*, along with their key bindings: See the section "Special Keys" in *Symbolics Common Lisp: Language Concepts*. We can use one of these options, or we can use other Debugger commands to examine or manipulate the stack. Let's use the `:Window Debugger (c-m-w)` command to invoke the Window Debugger.

Figure 7 shows the Window Debugger frame as it looks when we invoke it. The top window, an inspect pane, shows disassembled code for `compute-dens` with an arrow at the instruction that produced the error. The next window is an inspect history pane. The two windows side by side show the function's arguments and local variables and their values. The next window is a backtrace of the stack with an arrow at the frame that produced the error. The next window is a mouse-sensitive listing of options for proceeding or restarting. Next is a command menu. The bottom window is a Dynamic Lisp Listener with the error message displayed.

The disassembled code for `compute-dens` shows that the first argument to the subtraction that caused the error was the value of `*x2*`. We can inspect `*x2*` simply by clicking on its printed representation in the disassembled code.

Figure 8 shows the Window Debugger after we inspect `*x2*`. The value of `*x2*` is `nil`. We could have confirmed this by evaluating `*x2*` in the Lisp Listener pane.

Now, if we remember what the value of `*x2*` is supposed to be, we can set `*x2*` to that value by typing to the Lisp Listener pane:

```
(setq *x2* (- *p0x* *top-edge* *top-edge*))
```

We can then click on [Retry] to reinvoke the stack frame and continue the program.

If we forget the value of `*x2*`, we might want to look at the source code. We can

invoke the editor by clicking on [Edit] and then on the name of the function we want to edit. Inside the editor, we can change and recompile code. We can edit **draw-big-arrow** to bind **\*x2\*** and then recompile that function. If we entered the Debugger from the editor, we cannot return to the Debugger, but we can run the program again. Otherwise, we can return to the Window Debugger by pressing **c-Z** or **ABORT**. We can then set the value of **\*x2\*** and reinvoke the frame.

In the Debugger, **c-HELP** displays information on all Debugger commands. Following are some of the most useful commands:

---

### Reference

|                                    |                                                                       |
|------------------------------------|-----------------------------------------------------------------------|
| <b>:Show Argument (c-R)</b>        | Shows arguments for the current stack frame.                          |
| <b>:Edit Function (c-E)</b>        | Calls the editor to edit the function from the current frame.         |
| <b>:Show Frame (c-L)</b>           | Clears the screen and redisplay the original error message.           |
| <b>:Bottom Of Stack (m-&gt;)</b>   | Moves to the bottom of the stack and displays the least-recent frame. |
| <b>:Next Frame (c-N)</b>           | Moves down the stack by one frame.                                    |
| <b>:Previous Frame (c-P)</b>       | Moves up the stack by one frame.                                      |
| <b>:Top Of Stack (m-&lt;)</b>      | Moves to the bottom of the stack and displays the most recent frame.  |
| <b>:Return (c-R)</b>               | Returns a value from the current frame.                               |
| <b>:Show Backtrace (m-B)</b>       | Shows a backtrace of function names with arguments.                   |
| <b>:Show Local (c-m-L)</b>         | Shows local variables and disassembled code for the current frame.    |
| <b>:Reinvoke (c-m-R)</b>           | Reinvokes the current frame.                                          |
| <b>:Window Debugger (c-m-W)</b>    | Invokes the Window Debugger.                                          |
| <b>:Show Compiled Code (c-X D)</b> | Displays the disassembled code for a function.                        |



|                             |                                                                                                                 |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------|
| :Show Source Code (c-x c-D) | Displays the source code for a function.                                                                        |
| :Analyze Frame (c-m-Z)      | Analyzes the erroneous frame and locates the source code of the current error.                                  |
| :Describe Last (c-m-D)      | executes the Lisp <b>describe</b> function on the most recently displayed value and leaves * set to that value. |
| :Show Special               | Displays the special-variable binding of a symbol in the context of the current frame.                          |

### 30.3 Commenting Out Code

Sometimes a program runs but behaves in an unexpected way. In looking for the source of the problem, you might want to execute some portions of the program and disable others. An easy way to disable code without destroying it is to make a comment of it. You can comment out code by preceding it with a semicolon or surrounding it with #|...|#: See the section "Comments", page 249.

|                                                                                                                                                                                                                                                                                                         |         |        |         |       |     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|--------|---------|-------|-----|
| <i>More above</i>                                                                                                                                                                                                                                                                                       |         |        |         |       |     |
| <pre> COMPUTE-DENS  3 PUSH-INDIRECT *D1*  4 BUILTIN --INTERNAL STACK  5 PUSH-LOCAL FP 0 ;X  6 PUSH-INDIRECT *P0X*  7 BUILTIN --INTERNAL STACK 10 PUSH-INDIRECT *P2X* 11 PUSH-INDIRECT *P0X* =&gt; 12 BUILTIN --INTERNAL STACK 13 BUILTIN FLOAT STACK 14 BUILTIN /-INTERNAL STACK                 </pre> |         |        |         |       |     |
| <i>More below</i>                                                                                                                                                                                                                                                                                       |         |        |         |       |     |
| #<Stack-Frame COMPUTE-DENS PC=12>                                                                                                                                                                                                                                                                       |         |        |         |       |     |
| Args:<br>Arg 0 (X): 1800                                                                                                                                                                                                                                                                                |         |        | Locals: |       |     |
| <i>More above</i>                                                                                                                                                                                                                                                                                       |         |        |         |       |     |
| <pre> (DO-ARROW) (DRAW-ARROW-GRAPHIC 1200 1800 1800) (DRAW-BIG-ARROW) (STRIFE-ARROWHEAD) (COMPUTE-NLINES 1800) -&gt;(COMPUTE-DENS 1800)                 </pre>                                                                                                                                          |         |        |         |       |     |
| <i>More below</i>                                                                                                                                                                                                                                                                                       |         |        |         |       |     |
| Return to normal debugger, staying in error context.<br>Supply replacement argument<br>Return a value from the --INTERNAL instruction<br>Retry the --INTERNAL instruction<br>Lisp Top Level in Lisp Listener 1                                                                                          |         |        |         |       |     |
| What Error                                                                                                                                                                                                                                                                                              | Inspect | Return | Set arg | Retry | T   |
| Arglist                                                                                                                                                                                                                                                                                                 | Edit    | Throw  | Search  |       | NIL |
| >>Trap: The first argument given to SYS:--INTERNAL, NIL, was not a number.                                                                                                                                                                                                                              |         |        |         |       |     |

Choose a value by pointing at the value. Right gets object into error handler.  
 08/20/83 17:01:23 rom GRAPHICS: Tyl

Figure 7. The Window Debugger: inspecting the stack frame containing a call to compute-dens.

|                                                                                                                                                                                                               |         |        |                    |       |     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|--------|--------------------|-------|-----|
| <i>Top of object</i>                                                                                                                                                                                          |         |        |                    |       |     |
| <pre>*X2* Value is NIL Function is unbound Property list: (DOCUMENTATION "... SPECIAL #&lt;UNIX-PATHNAME "VIXEN: //dess//workstyles. Package: #&lt;Package GRAPHICS 36635277&gt;</pre>                        |         |        |                    |       |     |
| <i>Bottom of object</i>                                                                                                                                                                                       |         |        |                    |       |     |
| <pre>#&lt;Stack-Frame COMPUTE-DENS PC=12&gt; *X2*</pre>                                                                                                                                                       |         |        |                    |       |     |
| <pre>Args: Arg 0 (X): 1800</pre>                                                                                                                                                                              |         |        | <pre>Locals:</pre> |       |     |
| <i>More above</i>                                                                                                                                                                                             |         |        |                    |       |     |
| <pre>(DO-ARROW) (DRAW-ARROW-GRAPHIC 1200 1800 1800) (DRAW-BIG-ARROW) (STRIPE-ARROWHEAD) (COMPUTE-NLINES 1800) +(COMPUTE-DENS 1800)</pre>                                                                      |         |        |                    |       |     |
| <i>More below</i>                                                                                                                                                                                             |         |        |                    |       |     |
| <pre>Return to normal debugger, staying in error context. Supply replacement argument Return a value from the --INTERNAL instruction Retry the --INTERNAL instruction Lisp Top Level in Lisp Listener 1</pre> |         |        |                    |       |     |
| What Error                                                                                                                                                                                                    | Inspect | Return | Set arg            | Retry | T   |
| Arglist                                                                                                                                                                                                       | Edit    | Throw  | Search             |       | NIL |
| <pre>&gt;&gt;Trap: The first argument given to SYS:--INTERNAL, NIL, was not a number.</pre>                                                                                                                   |         |        |                    |       |     |

Choose a value by pointing at the value. Right gets object into error handler.  
08/20/83 17:02:05 rom GRAPHICS: Tyl

Figure 8. The Window Debugger: inspecting the variable \*x2\*.

**Example**

We have outlined the large arrow and the largest of the small arrows. We try to outline the rest of the small arrows by adding two recursive function calls to **do-arrows**:

```
(defun do-arrows ()
  ;; Don't exceed maximum recursion level
  (when (< *depth* *max-depth*)
    ;; Bind values for half and one-fourth of top edge
    (let ((*top-edge-2* (/ *top-edge* 2))
          (*top-edge-4* (/ *top-edge* 4)))
      (draw-arrow) ;Draw a small arrow
      ;; Increment depth. Divide top edge in half. Bind new
      ;; coordinates for top right point of next arrow.
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (+ *top-edge-4* (- *p0x* *top-edge*)))
            (*p0y* (- *p0y* *top-edge-2*)))
        ;; Draw a left-hand child arrow
        (do-arrows))
      ;; Increment depth. Divide top edge in half. Bind new
      ;; coordinates for top right point of next arrow.
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (- *p0x* *top-edge-2*))
            (*p0y* (+ *top-edge-4* (- *p0y* *top-edge*))))
        ;; Draw a right-hand child arrow
        (do-arrows))))))
```

This code produces the result shown in figure 9. Something is clearly wrong with at least one of the function calls, but the complexity of the figure makes it difficult to see the source of the error. We simplify the figure by making a comment of the second recursive function call:

```

(defun do-arrows ()
  ;; Don't exceed maximum recursion level
  (when (< *depth* *max-depth*)
    ;; Bind values for half and one-fourth of top edge
    (let ((*top-edge-2* (/ *top-edge* 2))
          (*top-edge-4* (/ *top-edge* 4)))
      (draw-arrow) ;Draw a small arrow
      ;; Increment depth. Divide top edge in half. Bind new
      ;; coordinates for top right point of next arrow.
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (+ *top-edge-4* (- *p0x* *top-edge*)))
            (*p0y* (- *p0y* *top-edge-2*)))
        ;; Draw a left-hand child arrow
        (do-arrows))
      ;; Increment depth. Divide top edge in half. Bind new
      ;; coordinates for top right point of next arrow.
      #||
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (- *p0x* *top-edge-2*)
                    (*p0y* (+ *top-edge-4* (- *p0y* *top-edge*))))
        ;; Draw a right-hand child arrow
        (do-arrows))))
    #||
    #||
  ))

```

We recompile `do-arrows` (using `c-sh-C`), run the program again, and obtain the results shown in figure 10. The small arrows now appear to be the right size, and the number of recursion levels is correct. The problem seems to lie in the positioning of the arrows, or the calculation of the new values for `*p0x*` and `*p0y*`. On close inspection, we see that the x-coordinates look correct, but the y-coordinates are wrong. Instead of obtaining the new value of `*p0y*` by subtracting `*top-edge-2*` from the old `*p0y*`, we should subtract `*top-edge-4*` from `*p0y*`. We change the definition of `do-arrows`:

```

(defun do-arrows ()
  .
  .
  (let ((*depth* (1+ *depth*))
        (*top-edge* *top-edge-2*)
        (*p0x* (+ *top-edge-4* (- *p0x* *top-edge*)))
        (*p0y* (- *p0y* *top-edge-4*)))
    ;; Draw a left-hand child arrow
    (do-arrows))
  ;; Increment depth. Divide top edge in half. Bind new
  ;; coordinates for top right point of next arrow.
  #||
  (let ((*depth* (1+ *depth*))
        (*top-edge* *top-edge-2*)
        (*p0x* (- *p0x* *top-edge-2*))
        (*p0y* (+ *top-edge-4* (- *p0y* *top-edge*)))
        ;; Draw a right-hand child arrow
        (do-arrows))))
  ||#
  )))

```

When we recompile `do-arrows` and run the program again, we obtain the results shown in figure 11. The first recursive function call is now correct. Looking at the arguments in the second function call, we see that the same error exists in the calculation of the new `*p0x*`: We should subtract `*top-edge-4*`, not `*top-edge-2*`, from the old `*p0x*`. We make the change, remove the `#||` and `||#`, and recompile `do-arrows`. We obtain the results shown in figure 2.

---

### Example

Figure 5 shows a split screen, with graphic output in the upper window and source code in the lower. To adjust the size of the graphic for the smaller window, we have to change the arguments to `draw-arrow-graphic` when we call that function from `do-arrow`. We want to keep a record of the arguments we use to produce a full-screen figure. We can make a comment of the call to `draw-arrow-graphic` that uses full-screen arguments:

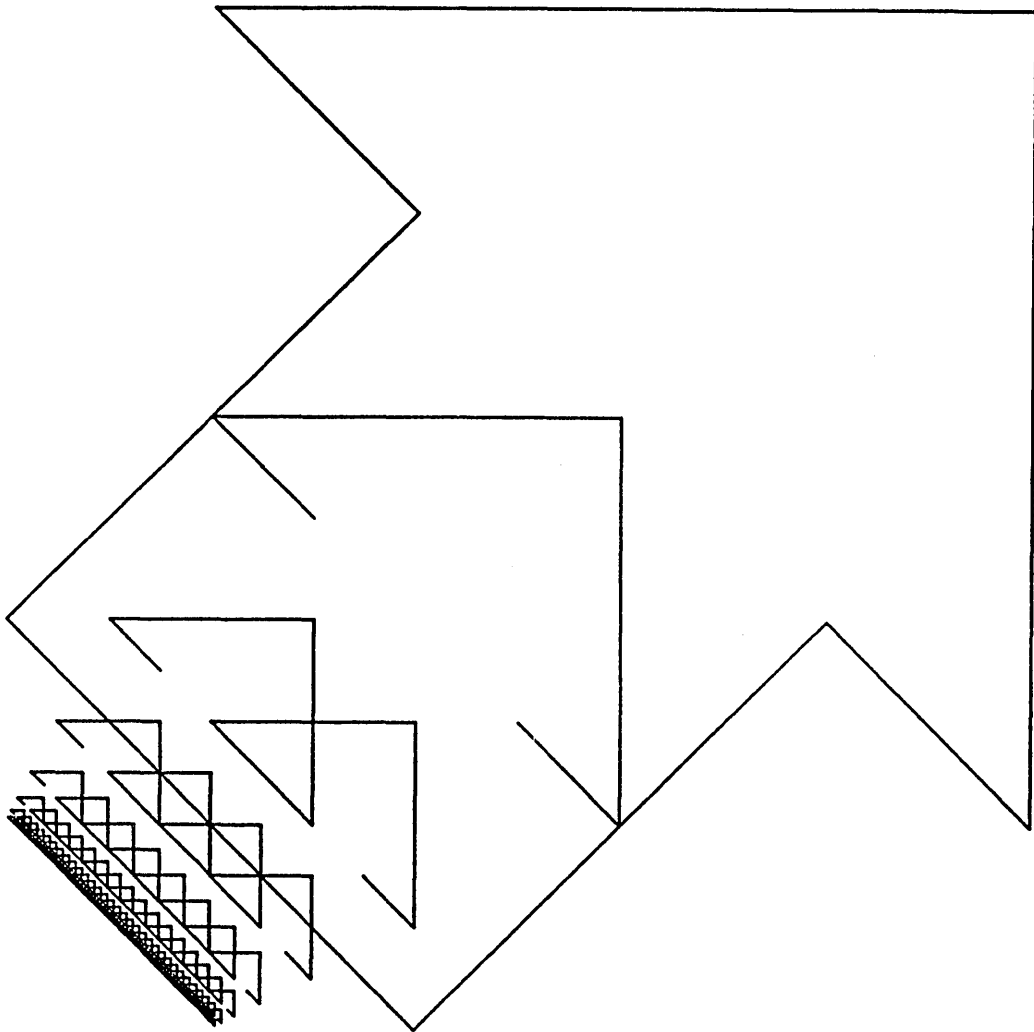


Figure 9. Output resulting from a faulty attempt to outline the small arrows recursively.

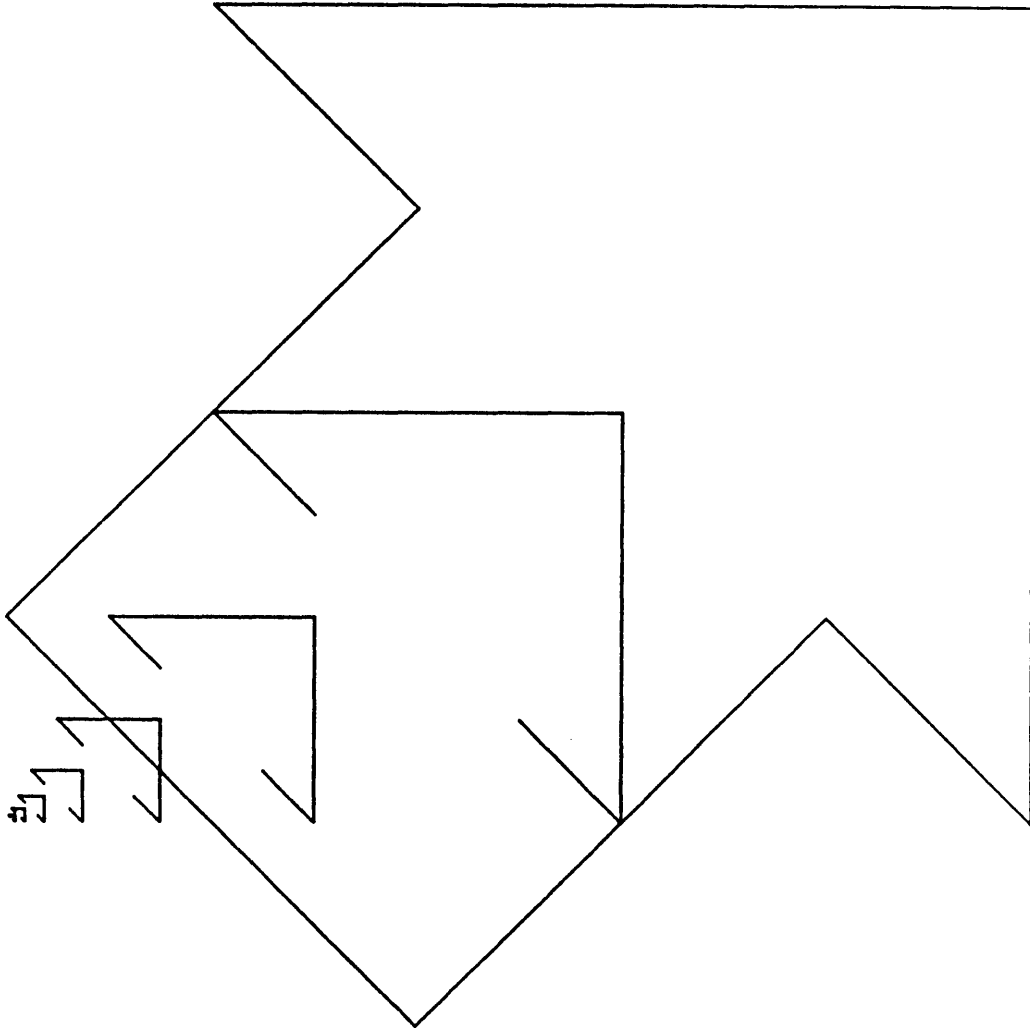


Figure 10. Output resulting from a faulty attempt to outline the small arrows recursively, with the second function call commented out.



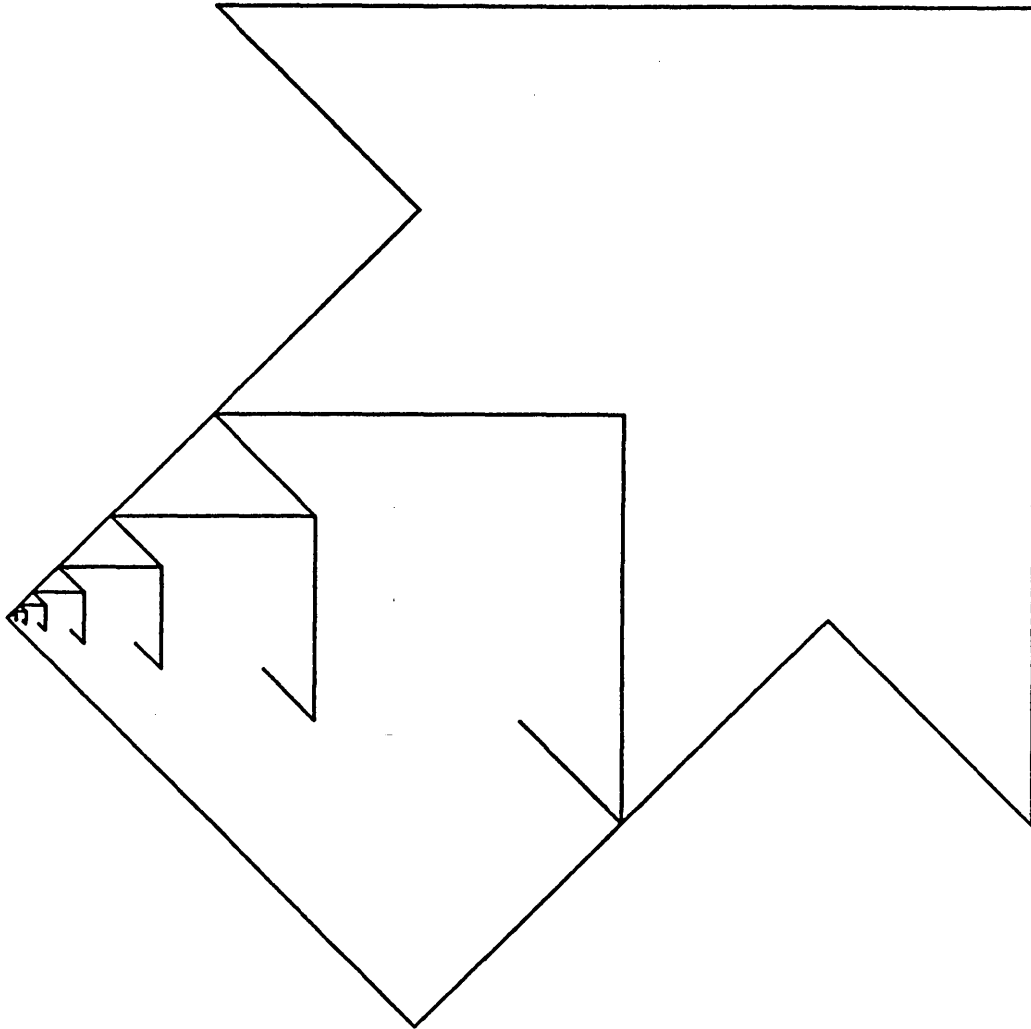


Figure 11. Output resulting from a corrected attempt to outline the small arrows recursively, with the second function call commented out.

```
(defun do-arrow ()
  (setq *dest* (make-instance 'screen-arrow-output))
  (send terminal-io ':clear-screen)
  ; (draw-arrow-graphic 1280 1800 1800))
  (draw-arrow-graphic 640 1300 1800))
```

## 30.4 Tracing and Stepping

### 30.4.1 Tracing

When a program runs but behaves unexpectedly, you might be calling functions in the wrong sequence or passing incorrect arguments. Tracing function calls can help detect this sort of problem. By default, tracing prints a message, indented according to the level of recursion, on entering and leaving a function. It also prints the arguments passed and the values returned.

You can invoke tracing in three ways:

- Use [Trace] in the System menu
- Use Trace (m-x) in Zmacs
- Use the `trace` special form

[Trace] and Trace (m-x) pop up a menu of options, including stepping and inserting breakpoints. You can use these options with `trace`, too, but the syntax is complex. Table 1 summarizes the correspondence between trace menu items and `trace` options. For a description of the options: See the section "Options To `trace`", page 74.

#### Example

Suppose that we had begun writing the recursive function calls in `do-arrows` with the following code, passing arguments to `do-arrows` instead of binding the special variables:

```
(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  .
  .
  (draw-big-arrow)
  (do-arrows 0 *top-edge-2* (- *p0x* *top-edge-2*) (- *p0y* *top-edge-2*)))
```

```

(defun do-arrows (*depth* *top-edge* *p0x* *p0y*)
  ;; Don't exceed maximum recursion level
  (when (< *depth* *max-depth*)
    ;; Bind new values for half and one-fourth of top edge
    (let ((*top-edge-2* (/ *top-edge* 2))
          (*top-edge-4* (/ *top-edge* 4)))
      ;; Draw a small arrow
      (draw-arrow)
      ;; Draw a left-hand child arrow, dividing top edge in half,
      ;; incrementing depth, and passing new coordinates for top
      ;; right point
      (do-arrows *top-edge-2* (1+ *depth*)
                  (+ *top-edge-4* (- *p0x* *top-edge*))
                  (- *p0y* *top-edge-4*))
      ;; Draw a right-hand child arrow, dividing top edge in half,
      ;; incrementing depth, and passing new coordinates for top
      ;; right point
      (do-arrows *top-edge-2* (1+ *depth*) (- *p0x* *top-edge-4*)
                  (+ *top-edge-4* (- *p0y* *top-edge*))))))

```

This code produces only the first of the small arrows. Again, something appears to be wrong with the recursive function calls. Using Trace (m-X), we trace calls to **do-arrows**. We run the program again, and the following trace output appears:

```

(1 ENTER DO-ARROWS (0 640 1160 1160))
(2 ENTER DO-ARROWS (320 1 680 1000))
(2 EXIT DO-ARROWS NIL)
(2 ENTER DO-ARROWS (320 1 1000 680))
(2 EXIT DO-ARROWS NIL)
(1 EXIT DO-ARROWS NIL)
NIL

```

The problem here is immediately apparent: The order of the first two arguments in the recursive function calls is reversed. We are passing the new value of **\*top-edge\*** as the new value of **\*depth\***. Because this value exceeds that of **\*max-depth\***, the function returns after the first recursive call.

---

## Reference

Trace (m-X)

Traces or untraces a specified

function. Prompts for the name of a function to trace and pops up a menu of trace options.

[Trace] (from the System menu) Traces or untraces a specified function. Prompts for the name of a function to trace and pops up a menu of trace options.

(trace (:function function-spec-1 option-1 option-2 ...) ...)  
 Enables tracing of one or more functions. If *function-spec* is a symbol, the keyword **:function** is unnecessary. An argument can also be a list whose car is a list of function names and whose cdr is one or more options. In this case, all functions in the list are traced with the same options. With no arguments, returns a list of functions being traced.

(untrace (:function function-spec-1) ...)  
 Disables tracing of one or more functions. If *function-spec* is a symbol, the keyword **:function** is unnecessary. With no arguments, untraces all functions being traced.

---

### 30.4.2 Stepping

When a program behaves unexpectedly and tracing doesn't reveal the problem, you might step through the evaluation of a function call. You can step through function execution by using **zl:step**, [Step] from a trace menu, or the **:step** option for **trace**.

You can step through the execution of a function only if it is interpreted, not compiled. If you want to step through execution of a compiled function, read the definition into a Zmacs buffer and use a Zmacs command (such as **c-sh-E**) to evaluate it. See the section "Evaluation and the Editor", page 303.

The Stepper prints a partial representation of each form evaluated and the values returned. A back arrow (←) precedes the representation of each form being evaluated. A double arrow (↔)

Table 1. Trace Menu Items and trace Options

| <i>Trace menu item</i> | <i>trace option</i>           | <i>Description</i>                                                                                    |
|------------------------|-------------------------------|-------------------------------------------------------------------------------------------------------|
| [Cond break before]    | <b>:break predicate</b>       | Enters breakpoint on function entry if <i>predicate</i> not nil                                       |
| [Break before]         | <b>:break t</b>               | Enters breakpoint on function entry                                                                   |
| [Cond break after]     | <b>:exitbreak predicate</b>   | Enters breakpoint on function exit if <i>predicate</i> not nil                                        |
| [Break after]          | <b>:exitbreak t</b>           | Enters breakpoint on function exit                                                                    |
| [Error]                | <b>:error</b>                 | Enters Debugger on function entry                                                                     |
| [Step]                 | <b>:step</b>                  | Steps through (interpreted) function execution                                                        |
| [Cond before]          | <b>:entrycond predicate</b>   | Prints trace output on function entry if <i>predicate</i> not nil                                     |
| [Cond after]           | <b>:exitcond predicate</b>    | Prints trace output on function exit if <i>predicate</i> not nil                                      |
| [Conditional]          | <b>:cond predicate</b>        | Prints trace output on function entry and exit if <i>predicate</i> not nil                            |
| [Print before]         | <b>:entryprint form</b>       | Prints value of <i>form</i> in trace entry output                                                     |
| [Print after]          | <b>:exitprint form</b>        | Prints value of <i>form</i> in trace exit output                                                      |
| [Print]                | <b>:print form</b>            | Prints value of <i>form</i> in trace entry and exit output                                            |
| [ARGPDL]               | <b>:argpdl pdl</b>            | On function entry, pushes list of function name and args onto <i>pdl</i> ; pops list on function exit |
| [Wherein]              | <b>:wherein function</b>      | Traces function only when called within <i>function</i>                                               |
| [Per Process]          | <b>:per-process process</b>   | Traces function only in <i>process</i>                                                                |
| [Untrace]              |                               | Calls <b>untrace</b> on function                                                                      |
|                        | <b>:entry list</b>            | Prints values of forms in <i>list</i> on function entry                                               |
|                        | <b>:exit list</b>             | Prints values of forms in <i>list</i> on function exit                                                |
|                        | <b>:arg :value :both :nil</b> | Controls printing of args on function entry and values on function exit                               |

precedes macro forms. A forward arrow (→) precedes returned values.

After printing, the Stepper waits for a command before proceeding to the next step. Stepper commands allow you to specify the level of evaluation to be stepped, escape to the editor, or enter a Lisp breakpoint loop. For a list of commands, press HELP inside the Stepper, or: See the section "Stepping Through an Evaluation", page 85. Following are some basic Stepper commands:

| <i>Command</i> | <i>Action</i>                                                                                |
|----------------|----------------------------------------------------------------------------------------------|
| c-N            | Evaluate until next thing to print                                                           |
| SPACE          | Evaluate until next thing to print at this level (don't step at lower levels)                |
| c-U            | Evaluate until next thing to print at next level up (don't step at current and lower levels) |
| c-B            | Enter breakpoint loop                                                                        |
| c-E            | Enter Zmacs                                                                                  |
| c-X            | Evaluate until finished (exit from stepping)                                                 |

---

### Example

We have the same problem with the function **do-arrows** as we described elsewhere: See the section "Tracing", page 323. The program outlines only the largest of the small arrows, indicating a problem with the recursive function calls. Instead of just tracing **do-arrows**, we step through its evaluation. We first use c-sh-E to evaluate the definition of **do-arrows**. We then use [Step] in the menu that Trace (m-X) pops up to trace and step through **do-arrows**. We run the program. The Stepper waits for a command before evaluating each form in **do-arrows**. We press SPACE to skip to the next form at the same level. When we come to the comparison of **\*depth\*** and **\*max-depth\*** in the recursive calls, we want to see each level of evaluation. We press c-N at each of these steps. The tracing and stepping output looks as follows:

```

(1 ENTER DO-ARROWS (0 640 1160 1160))
↔ (WHEN (< *DEPTH* *MAX-DEPTH*) (LET ((*TOP-EDGE-2* (/ *TOP-EDGE*
← (COND ((< *DEPTH* *MAX-DEPTH*) (PROGN (LET ((*TOP-EDGE-2* (/ *T
(2 ENTER DO-ARROWS (320 1 680 1000))
↔ (WHEN (< *DEPTH* *MAX-DEPTH*) (LET ((*TOP-EDGE-2* (/ *TOP-EDGE*
← (COND ((< *DEPTH* *MAX-DEPTH*) (PROGN (LET ((*TOP-EDGE-2* (/ *T
← (< *DEPTH* *MAX-DEPTH*)
← *DEPTH* → 320
← *MAX-DEPTH* → 7
← (< *DEPTH* *MAX-DEPTH*) → NIL
← (COND ((< *DEPTH* *MAX-DEPTH*) (PROGN (LET ((*TOP-EDGE-2* (/ *T → NIL
(2 EXIT DO-ARROWS NIL)
(2 ENTER DO-ARROWS (320 1 1000 680))
↔ (WHEN (< *DEPTH* *MAX-DEPTH*) (LET ((*TOP-EDGE-2* (/ *TOP-EDGE*
← (COND ((< *DEPTH* *MAX-DEPTH*) (PROGN (LET ((*TOP-EDGE-2* (/ *T
← (< *DEPTH* *MAX-DEPTH*)
← *DEPTH* → 320
← *MAX-DEPTH* → 7
← (< *DEPTH* *MAX-DEPTH*) → NIL
← (COND ((< *DEPTH* *MAX-DEPTH*) (PROGN (LET ((*TOP-EDGE-2* (/ *T → NIL
(2 EXIT DO-ARROWS NIL)
(1 EXIT DO-ARROWS NIL)
NIL

```

In this example, stepping shows even more clearly than tracing that the value of *\*depth\** is wrong in the recursive function calls.

---

### Reference

|                                                |                                                                                                                                           |
|------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| (zl:step <i>form</i> )                         | Steps through the evaluation of <i>form</i>                                                                                               |
| Trace (n-X) [Step]                             | Steps through the execution of a function being traced.                                                                                   |
| [Trace / Step] (from the System menu)          | Steps through the execution of a function being traced.                                                                                   |
| (trace (:function <i>function-spec</i> :step)) | Steps through the execution of a function being traced. If <i>function-spec</i> is a symbol, the keyword <b>:function</b> is unnecessary. |

## 30.5 Using Breakpoints

In debugging a program, you might want to interrupt function execution to enter a Lisp breakpoint loop or the Debugger. Entering the Debugger is usually more useful, for there you can examine the stack, return values, and take other steps in addition to evaluating forms.

You can use two general kinds of breakpoints:

- You can edit into a definition a call to **zl:dbg** (with no arguments) or to **zl:break**. The advantage of this kind of breakpoint is that, as with stepping, you can interrupt execution within the function. The disadvantage is that you have to edit and recompile the definition to insert and remove the breakpoint. If you redefine the function after inserting the breakpoint, the breakpoint might be lost.
- You can use **breakon** or one of the error or break options to **trace**. These features create *encapsulations*, functions that contain the *basic definitions* of the functions to which you want to add breakpoints. For more on encapsulations: See the section "Encapsulations" in *Symbolics Common Lisp: Language Concepts*. The advantage of this kind of breakpoint is that when you recompile or otherwise redefine the function, only the basic definition is replaced, and the breakpoints remain. The disadvantage is that you can interrupt function execution only on entry or exit, not within the function.

You insert these breakpoints by calling **breakon** or **trace** from a Lisp Listener or by using the trace menu; you remove them by calling **unbreakon** or **untrace**. When you break on entering function execution, just before applying the function to its arguments, the variable **arglist** is bound to a list of the arguments. When you break on exiting from function execution, just before the function returns, the variable **values** is bound to a list of the returned values.

From either a breakpoint loop or the Debugger, **RESUME** allows the program to continue, and **ABORT** returns control to the previous break or, if none exists, to top level.

---

### Example

We decide to break on entry to **do-arrows** and enter the Debugger while tracing the function. We use **Trace (M-X)** and then **[Error]** from the trace menu. We select a Lisp Listener and run the program. On the first entry to **do-arrows** we enter the Debugger, with the following message:



>> TRACE Break: DO-ARROWS entered.

DO-ARROWS: (encapsulated for TRACE)  
 Rest arg (ARGLIST): (0 640 1160 1160)  
 s-A, RESUME: Proceed without any special action  
 s-B, ABORT: Lisp Top Level in Lisp Listener 1  
 →

---

### Reference

- (zl:dbg *process*)** Enters the Debugger in *process*. With an argument of *t*, finds a process that has sent an error notification. With no argument, enters the Debugger as if an error had occurred in the current process.
- (zl:break *tag conditional-form*)** Enters a Lisp breakpoint loop (identified as "breakpoint *tag*") if *conditional-form* is not *nil* or is not supplied.
- (breakon *function-spec conditional-form*)** Passes control to the Debugger on entering *function-spec* if *conditional-form* is not *nil* or is not supplied. With no arguments, returns a list of functions with breakpoints specified by **breakon**.
- (unbreakon *function-spec conditional-form*)** Turns off the breakpoint condition specified by *conditional-form* for *function-spec*. If *conditional-form* is not supplied, turns off all breakpoints specified by **breakon** for *function-spec*. With no arguments, turns off all breakpoints specified by **breakon** for all functions.

- [Error] (from a trace menu) Passes control to the Debugger on entering a function being traced.
- [Cond break before] (from a trace menu) Prompts for a predicate. Displays trace entry information and enters a Lisp breakpoint loop on entering a function being traced if the predicate is not `nil`.
- [Cond break after] (from a trace menu) Prompts for a predicate. Displays trace exit information and enters a Lisp breakpoint loop on exiting from a function being traced if the predicate is not `nil`.
- (trace (:function *function-spec* :error))**  
Passes control to the Debugger on entering a function being traced. If *function-spec* is a symbol, the keyword `:function` is unnecessary.
- (trace (:function *function-spec* :break *predicate*))**  
Prints trace entry information and, if the value of *predicate* is not `nil`, enters a Lisp break loop on entering the function. If *function-spec* is a symbol, the keyword `:function` is unnecessary.
- (trace (:function *function-spec* :exitbreak *predicate*))**  
Prints trace exit information and, if the value of *predicate* is not `nil`, enters a Lisp break loop on exiting from the function. If *function-spec* is a symbol, the keyword `:function` is unnecessary.

## 30.6 Expanding Macros

Sometimes a program bug appears to stem from unexpected behavior by a macro. Seeing how a macro form expands can help find the bug. To be sure that a macro does what you want it to, you might also want to create and expand a macro form soon after defining the macro and compiling the definition.

You can expand a macro form in a Zmacs buffer using Macro Expand Expression (`c-sh-M`). This command expands the form following point, but not any macro forms within it. To expand all subforms, use Macro Expand Expression All (`m-sh-M`). You can also expand macro forms with `mexp`, which enters a loop to read and expand one form after another.

---

### Example

We have just written code to stripe the shafts of the small arrows, drawing stripes with uniform spacing and density. We produce the results shown in figure 12. We evidently have a problem with the function `draw-arrow-shaft-stripes`. The code for this function is as follows:

```
(defun draw-arrow-shaft-stripes
  (left-x top-y right-x bottom-y)
  ;; Find y-coord of starting point of stripe. Don't go
  ;; below the bottom of the triangle.
  (loop for start-y from top-y by *stripe-distance* above bottom-y
        ;; Find x-coord of ending point of the stripe
        for end-x from right-x by *stripe-distance*
        ;; Draw a stripe
        do (draw-arrow-shaft-lines
           left-x start-y end-x bottom-y)))
```

The bug stems from incorrect coordinates for the endpoints of the shaft stripes. The beginning coordinates (`left-x` and `start-y`) are correct. The ending y-coordinate (`bottom-y`) looks right, but the ending x-coordinate (`end-x`) is wrong. The problem might not be evident from looking at the code, which consists entirely of a `zl:loop` form. We move to the beginning of the `zl:loop` form and expand it, using `c-sh-M`:

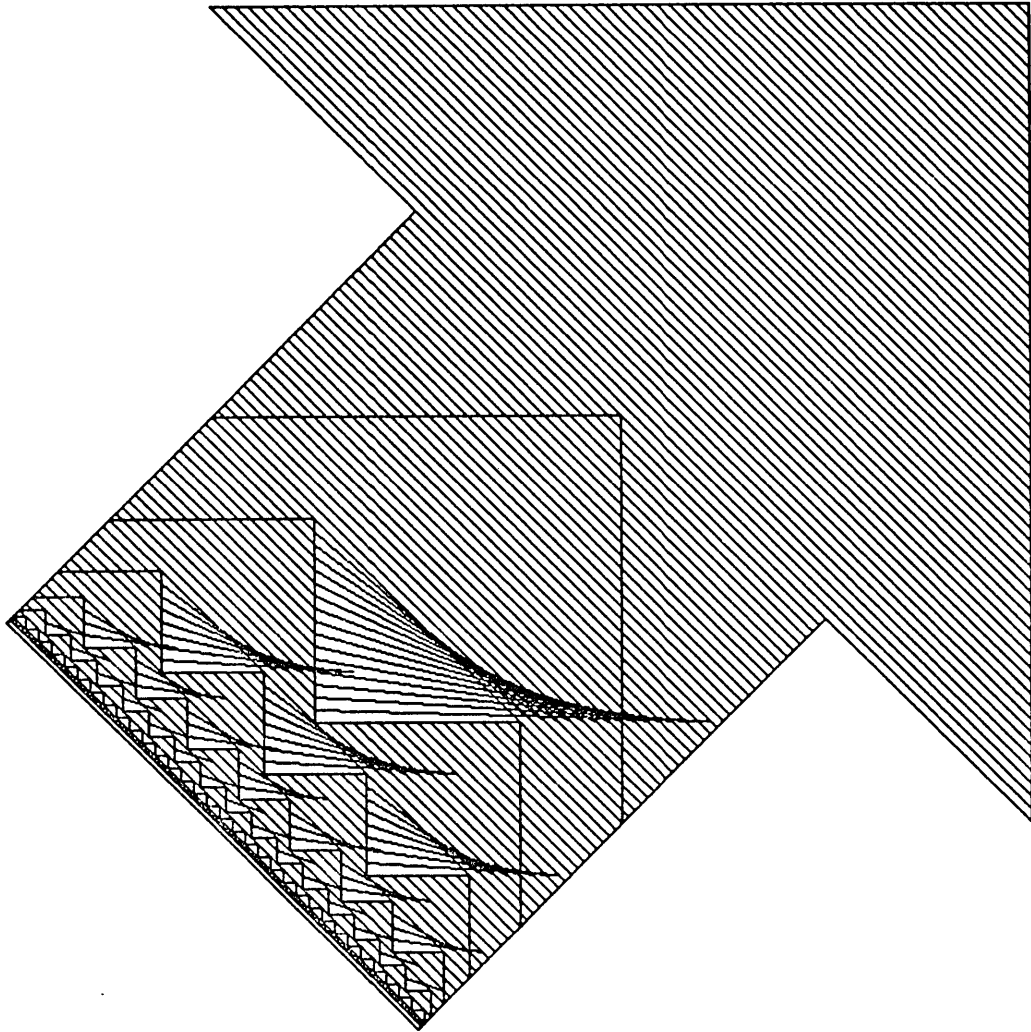


Figure 12. Output from the program with a bug in the function `draw-arrow-shaft-stripes`.

```

((LAMBDA (START-Y G1049 G1050)
  ((LAMBDA (END-X G1051)
    (PROG NIL
      (AND (NOT (GREATERP START-Y G1050)) (GO SI:END-LOOP))
      SI:NEXT-LOOP
      (DRAW-ARROW-SHAFT-LINES LEFT-X START-Y END-X BOTTOM-Y)
      (SETQ START-Y (DIFFERENCE START-Y G1049))
      (AND (NOT (GREATERP START-Y G1050)) (GO SI:END-LOOP))
      (SETQ END-X (PLUS END-X G1051))
      (GO SI:NEXT-LOOP)
      SI:END-LOOP
    ))
  RIGHT-X
  *STRIPE-DISTANCE*))
TOP-Y
*STRIPE-DISTANCE*
BOTTOM-Y)

```

The expansion shows the **lambda**-bindings and **prog** form that the **zl:loop** macro creates. We can see that the error is in the setting of **end-x** within the **prog** form: We are incrementing **end-x** by **\*stripe-distance\***, when we should be decrementing it. The problem is in our use of a **zl:loop** keyword. Instead of writing

```
for end-x from right-x by *stripe-distance*
```

we should have written

```
for end-x downfrom right-x by *stripe-distance*
```

We make the change and recompile **draw-arrow-shaft-stripes**. Now if we expand the **zl:loop** form, we see that we are decrementing **end-x**:

```

((LAMBDA (START-Y G1062 G1063)
  ((LAMBDA (END-X G1064)
    (PROG NIL
      (AND (NOT (GREATERP START-Y G1063)) (GO SI:END-LOOP))
      SI:NEXT-LOOP
      (DRAW-ARROW-SHAFT-LINES LEFT-X START-Y END-X BOTTOM-Y)
      (SETQ START-Y (DIFFERENCE START-Y G1062))
      (AND (NOT (GREATERP START-Y G1063)) (GO SI:END-LOOP))
      (SETQ END-X (DIFFERENCE END-X G1064))
      (GO SI:NEXT-LOOP)
      SI:END-LOOP
    ))
  RIGHT-X
  *STRIPE-DISTANCE*))
TOP-Y
*STRIPE-DISTANCE*
BOTTOM-Y)

```

---

### Reference

#### Macro Expand Expression (c-sh-M)

Expands the macro form following point. Does not expand subforms within the form.

#### Macro Expand Expression All (m-sh-M)

Expands the macro form following point and all subforms within the form.

#### (mexp)

Enters a loop: prompts for a macro form to expand, expands it, and prompts for another macro form. Exits from the loop on **nil**.

## 30.7 Using the Inspector

The Inspector is a window-based tool that combines the **describe** and **disassemble** functions. Invoke it with **inspect**, **SELECT I**, or **[Inspect]** from the System menu. If you use **inspect**, the Inspector is not a separate activity from the Lisp Listener in which you invoke it. In that case you cannot use **SELECT L** to return to the Lisp Listener; you must click on **[Exit]** or **[Return]** in the Inspector menu.

The Inspector displays information about an object and lets you modify the object. It displays information for the last object inspected in the bottom window. It displays information for the two previous objects in the windows above the bottom one. It maintains a mouse-sensitive listing of all inspected objects in the history window. These are some of its useful features:

- The information the Inspector displays depends on the object's type. For a symbol, it displays a representation of the value, function, property list, and package. For a symbol's flavor property, it displays information about instance variables, component and dependent flavors, the message handler, init keywords, and the flavor property list. For a compiled function, it displays the disassembled assembly-language code that represents the compiler output.
- The Inspector is especially useful for examining data structures. It displays the names and values of the slots of structures and, unlike `describe`, the elements of (one-dimensional) arrays. For instances of flavors, the Inspector displays the names and values of instance variables.
- Within each display, most representations of objects are mouse sensitive. If you click on an object representation, you inspect that object. For example, you can inspect elements of lists. If an element of an array is itself an array, you can inspect the second array. In this way you can follow long paths in data structures.
- You can change a value by using the [Modify] option in the Inspector's menu. You can return a value when you exit the Inspector by clicking on [Return].

For more on the Inspector: See the section "The Inspector", page 93.

---

### Example

Suppose we had represented each arrow as an instance of a structure (defined with `zl:defstruct`) instead of a collection of special-variable values. We could have called the structure representing the small arrows `arrow` and set the value of a special variable, `*arr*`, to each instance of the structure as we created it.

Figure 13 shows an Inspector window for the last arrow in the figure. We first run the program in a Lisp Listener, then invoke the Inspector using `SELECT I`. Because we typed `(pkg-goto 'graphics)` in the Lisp Listener, the Inspector's package is `graphics`. We type `*arr*` to the interaction pane at the top of the

frame. The window at the bottom of the frame displays the names and values of the structure slots. We can change these values by using the [Modify] menu option.

---

### Example

Suppose we had represented each arrow as an instance of a flavor and defined most of our computation functions as flavor methods instead of simple functions. We could have called the flavor representing the small arrows **arrow** and set the value of **\*arr\*** to each instance of the flavor as we created it.

Figure 14 shows an Inspector window for the last arrow in the figure. As with our structure example, we first run the program and then invoke the Inspector to evaluate **\*arr\*** and inspect the flavor instance that is its value. The Inspector displays the names and values of instance variables and a representation of the flavor's message handler.

We next click on the mouse-sensitive representation of the message handler. The Inspector displays a representation of the function spec for the method that handles each message. If we click on the function spec for the **:compute-dens** method of flavor **basic-arrow**, the Inspector displays the method's disassembled code.

---

### Reference

|                                  |                                                                                                                             |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| ( <i>inspect object</i> )        | Selects an Inspector window in which to inspect <i>object</i> .                                                             |
| SELECT I                         | Selects an Inspector window.                                                                                                |
| [Inspect] (from the System menu) | Selects an Inspector window.                                                                                                |
| ( <i>disassemble function</i> )  | Prints a representation of the assembly-language instructions for a compiled function.                                      |
| Disassemble (n-X)                | Prompts for the name of a compiled function and displays a representation of the function's assembly-language instructions. |



|                                                                                                                                                                                                                                                                          |                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| *arr*                                                                                                                                                                                                                                                                    |                                                                                |
| #<ARROW -33247021>                                                                                                                                                                                                                                                       | <i>Top of History</i><br>Exit<br>Return<br>Modify<br>DeCache<br>Clear<br>Set \ |
| <i>Bottom of History</i>                                                                                                                                                                                                                                                 |                                                                                |
| <i>Top of object</i>                                                                                                                                                                                                                                                     |                                                                                |
| Empty                                                                                                                                                                                                                                                                    |                                                                                |
| <i>Bottom of object</i>                                                                                                                                                                                                                                                  |                                                                                |
| <i>Top of object</i>                                                                                                                                                                                                                                                     |                                                                                |
| Empty                                                                                                                                                                                                                                                                    |                                                                                |
| <i>Bottom of object</i>                                                                                                                                                                                                                                                  |                                                                                |
| <i>Top of object</i>                                                                                                                                                                                                                                                     |                                                                                |
| #<ARROW -33247021><br>Named structure of type ARROW<br><br>DEPTH: 6<br>TOP-EDGE: 10<br>TOP-EDGE-2: 5<br>TOP-EDGE-4: 2<br>X2: 825<br>STRIPE-D: 10<br>P0X: 845<br>P0Y: 215<br>P1X: 835<br>P1Y: 215<br>P2X: 837<br>P2Y: 213<br>P5X: 843<br>P5Y: 207<br>P6X: 845<br>P6Y: 205 |                                                                                |
| <i>Bottom of object</i>                                                                                                                                                                                                                                                  |                                                                                |

Choose a value by pointing at the value. Right finds function definition.  
08/17/83 18:23:32 rom GRAPHICS: Ty1\_\_

Figure 13. The Inspector window: inspecting an instance of a structure.

#arrt  
 #<ARROW 10020042> *Top of History* Exit  
Return  
Modify  
DeCache  
Clear  
Set \  
*Bottom of History*  
*Top of object*  
 Empty  
*Bottom of object*  
*Top of object*  
 Empty  
*Bottom of object*  
*Top of object*  
 #<ARROW 10020042>  
 An instance of ARROW. #<Message handler for ARROW>  
 DEPTH: 6  
 TOP-EDGE: 10  
 TOP-EDGE-2: 5  
 TOP-EDGE-4: 2  
 X2: 825  
 STRIPE-D: 10  
 P0X: 845  
 P0Y: 215  
 P1X: 835  
 P1Y: 215  
 P2X: 837  
 P2Y: 213  
 P5X: 843  
 P5Y: 207  
 P6X: 845  
 P6Y: 205  
*Bottom of object*

Choose a value by pointing at the value. Right finds function definition.  
 08/20/83 17:09:18 rom GRAPHICS: Tyl

Figure 14. The Inspector window: inspecting an instance of a flavor.







## 31. Using Flavors and Windows

All Genera programmers must know how to use flavors and the window system in at least an elementary way. Flavors are the basis of a powerful, nonhierarchical kind of object-oriented programming. Even if you don't use them extensively, the system code does. Applications that include screen display or user interaction must deal with the window system, which is itself built on flavors.

In this chapter we present a brief introduction to using flavors and windows. We do not discuss the concepts and organization of flavors and the window system in any detail. Instead, we modify the output module of our example program to show some simple uses of flavors, windows, and menus. We show basic examples of the following features:

- Using base, mixin, and instantiable flavors and **:daemon** method combination
- Creating a simple window and associating it with a process
- Producing LGP output
- Altering values using a choose-variable-values window
- Signalling a condition and proceeding

We also present some editor commands and Lisp functions for finding information about flavors and windows. Among the issues we do *not* discuss in any detail are the following:

- Using types of method combination other than **:daemon**
- Interacting with the mouse process
- Creating frames
- Specifying fonts
- Using menus

For more information on flavors and windows, read the following:

- On flavors: See the section "Flavors" in *Symbolics Common Lisp: Language Concepts*.
- On windows: See the section "Using the Window System" in *Programming the User Interface, Volume B*.
- On menus: See the section "Window System Choice Facilities" in *Programming the User Interface, Volume B*.
- On conditions and errors: See the section "Conditions" in *Symbolics Common Lisp: Language Concepts*.

### 31.1 Program Development: Modifying the Output Module

As now written, the output routines of our example program consist of a flavor and methods that produce lines on the stream to which `zl:terminal-io` is bound:

```
(deflavor screen-arrow-output
  ((scale-factor 2.5))
  ())

(defmethod (screen-arrow-output :show-lines)
  (x y &rest x-y-pairs)
  (loop for x0 = (send self ':compute-x x) then x1
        for y0 = (send self ':compute-y y) then y1
        for (x1 y1) on x-y-pairs by #'cddr
        do (setq x1 (send self ':compute-x x1)
                y1 (send self ':compute-y y1))
           (send terminal-io ':draw-line
                  x0 y0 x1 y1 tv:alu-ior t)))

(defmethod (screen-arrow-output :compute-x) (x)
  (fixr (/ x scale-factor)))

(defmethod (screen-arrow-output :compute-y) (y)
  (fixr (- 800 (/ y scale-factor))))
```

We want to be able to produce output on the screen, an LGP, or a file. For this we need a simple device-independent graphics system that uses *generic operations*. The central operation is `:show-lines`, which receives endpoint coordinates from the calculation module and produces lines on the appropriate output stream. Our general strategy for creating the output options is as follows:

1. Define a flavor and methods to calculate the position of the arrow figure on the screen or page. We can use this mixin with flavors that produce any kind of output.
2. Define flavors and methods to produce screen output. We build the instantiable flavors on `tv:window` and instantiate them with `tv:make-window`. We define two kinds of arrow window flavors:
  - A basic flavor that performs output and redisplay the window after changes.
  - A flavor, which we instantiate, that is built on the basic window and includes a mixin to convert LGP coordinates to screen coordinates.

3. Define a flavor and methods to produce LGP or file output.
4. Define a top-level function that uses a choose-variable-values window to select the type of output and alter some variables. The function calls `tv:make-window` or makes an instance of the LGP flavor, depending on the output type.
5. Change the arrow-window flavors to allow multiple windows, associate each window with its own process, and allow the user to modify the characteristics of the figure in each window.
6. Define a function to check for mistakes when the user changes the values of variables. We define condition flavors for the incorrect choices. We define handlers for the conditions and use `signal` to signal them. We allow the user to proceed by supplying new values for the variables.

We want to preserve modularity in writing these new routines. We define the flavor that positions the arrow figure so that we can use it with any sort of output. We keep the operations that transform LGP to screen coordinates separate from the basic window operations. We define the routines that handle bad variable values as separate flavors and functions. These precautions make it easy to define new kinds of windows or to check for errors in other variable values in the future.

---

### 31.1.1 A Mixin to Position the Figure

No matter what the output device, we want to be sure that the figure fits within the bounds of the page or window and is centered within the page or window. We define a mixin flavor, `arrow-parameter-mixin`, with methods to perform these calculations. We include this flavor in all flavors that produce output for the figure.

We define five instance variables to hold the parameters. Three of these, `top-edge`, `right-x`, and `top-y`, are the arguments we must pass to the calculation module. We make these three instance variables `gettable` so that we can retrieve them by sending messages to an instance of the dependent flavor. The other two instance variables are the width and height of the page or window in the appropriate units, either LGP or screen pixels.



```

| (def flavor arrow-parameter-mixin
|   (width height top-edge right-x top-y)
|   ()
|   (:gettable-instance-variables top-edge right-x top-y)
|   (:documentation :mixin
|     "Provides parameters for size and position of figure.
| Instance variables hold width and height of page or window;
| length of top edge of figure; coordinates of top right point
| of figure."))

```

The task of this flavor is to perform a generic operation, which we call **:compute-parameters**. This operation consists of separate computations for **top-edge**, **right-x**, and **top-y**. We define primary methods for these operations here, using coordinates with the origin at bottom left. Flavors that mix in this one can add daemons, whoppers, or their own primary methods to accommodate other coordinate systems and scale factors.

We perform these operations as follows:

1. Determine the width and height of the page or window. The details of this operation are the business of other flavors. We specify a required method, **:compute-width-and-height**, for any flavor that mixes in this one. We send **self** a **:compute-width-and-height** message to set the instance variables.
2. Calculate a provisional value for **top-edge** so that the figure fits within the smaller dimension of the page or window. We allow the user to specify, by setting the global variable **\*fill-proportion\***, what fraction of this dimension the figure should fill.
3. Adjust the top edge so that its value is at least 128 and is a multiple of 128 if larger. This adjustment ensures that stripe spacing is continuous throughout the levels of the figure.
4. Calculate **right-x** and **top-y** so that we center the figure within the page or window.

The complete code for this flavor and its methods is as follows:

```

| (defvar *fill-proportion* 0.9
|   "Proportion of smaller dimension to be filled by figure")

```

```

(defflavor arrow-parameter-mixin
  (width height top-edge right-x top-y)
  ()
  (:gettable-instance-variables top-edge right-x top-y)
  (:required-methods :compute-width-and-height)
  (:documentation :mixin
    "Provides parameters for size and position of figure.
    Instance variables hold width and height of page or window;
    length of top edge of figure; coordinates of top right point
    of figure. Methods calculate size and position of figure by
    centering it within the page or window and making it fill no
    more than the specified proportion of the smaller dimension.
    The methods use a coordinate system with origin at bottom left;
    other mixins must correct for this if output is going to a
    window. Other flavors must also provide a method for calculating
    width and height of the page or window. This flavor should be
    mixed into any instantiable flavor that produces output for the
    arrow graphic."))

  ;;; Method controlling calculation of size and position of figure.
  ;;; Sends messages to self to calculate width and height of page
  ;;; or window, length of top edge of figure, and coordinates of
  ;;; figure's top right point. These are separate methods so that
  ;;; other flavors can shadow them or add daemons. Another flavor
  ;;; must provide a method to compute width and height, because
  ;;; this is specific to the output device.
  (defmethod (arrow-parameter-mixin :compute-parameters) ()
    ;; Another flavor must supply method for width and height
    (send self ':compute-width-and-height)
    ;; Make a preliminary estimate of length of top edge
    (send self ':compute-top-edge)
    ;; Adjust top edge to make it a multiple of 128
    (send self ':adjust-top-edge)
    ;; Calculate coordinates of top right point of figure.
    ;; We can't do this until we know how long top edge is.
    (send self ':compute-right-x)
    (send self ':compute-top-y))

```

```

|   ;;; Makes a preliminary estimate of length of top edge.
|   ;;; The top edge of the arrow is 80 percent of the horizontal
|   ;;; or vertical length of the whole figure. First finds the
|   ;;; smaller of the length or width of the page or window.
|   ;;; Multiplies this by the proportion of this dimension that
|   ;;; is to be filled by the figure. The result is the
|   ;;; horizontal or vertical length of the figure. Multiplies
|   ;;; this by 0.8 to get the length of the top edge.
|   (defmethod (arrow-parameter-mixin :compute-top-edge) ()
|     (setq top-edge
|           (fixr (* 0.8 *fill-proportion* (min width height))))))

|   ;;; Adjusts length of top edge so it is a multiple of 128.
|   ;;; There are 64 stripes in the head of the large arrow. The
|   ;;; calculation module divides the length of top edge by two
|   ;;; each time it goes down another recursion level. By making
|   ;;; the original top edge a multiple of 128, we maximize
|   ;;; continuity in striping between arrowheads and shafts and
|   ;;; among the first several levels of recursion.
|   (defmethod (arrow-parameter-mixin :adjust-top-edge) ()
|     (setq top-edge
|           ;; Minimum length of top edge is 128
|           (if (< top-edge 256) 128
|               ;; Otherwise set to next lower multiple of 128
|               (* 128 (fix (// top-edge 128)))))

|   ;;; Calculates x-coordinate of top right point of figure.
|   ;;; Finds horizontal length of figure by dividing length of
|   ;;; top edge by 0.8. Centers the figure horizontally within
|   ;;; the page or window.
|   (defmethod (arrow-parameter-mixin :compute-right-x) ()
|     (setq right-x
|           (fixr (* 0.5 (+ width (// top-edge 0.8)))))

|   ;;; Calculates y-coordinate of top right point of figure.
|   ;;; Assumes that the origin is at bottom. Finds vertical
|   ;;; length of figure by dividing length of top edge by 0.8.
|   ;;; Centers the figure vertically within the page or window.
|   (defmethod (arrow-parameter-mixin :compute-top-y) ()
|     (setq top-y
|           (fixr (* 0.5 (+ height (// top-edge 0.8)))))

```

### 31.1.2 The Basic Arrow Window

We want to build our window on `tv:window`, a flavor that produces a simple window with borders, a label, and graphics. Any arrow window we use must provide for initialization and redisplay, determine its width and height, and supply a `:show-lines` method to draw our figure.

We define a mixin flavor, `basic-arrow-window-mixin`, with methods to do these things. We require that this flavor be used with `arrow-parameter-mixin` and `tv:window`. For the basic window, we assume that the coordinates supplied to `:show-lines` are screen coordinates, with origin at top left.

We write `basic-arrow-window-mixin` as follows:

1. Define the flavor. The `:required-flavors` option ensures that we have access to the flavors' instance variables and that an error will be signalled if someone makes an instance of a flavor that includes `basic-arrow-window-mixin` but not the required flavors. The `:default-init-plist` option provides values for some elements of the initialization property list in case no one else specifies them. The `:edges-from` option with an argument of `:mouse` allows the user to specify the initial size and position of the window by using mouse corners. We give an initial minimum width and height for the window because the length of `top-edge` must be at least 128, and we want the entire figure to fit inside the window.

```
| (defflavor basic-arrow-window-mixin () ()
|   (:required-flavors arrow-parameter-mixin tv:window)
|   (:default-init-plist
|     :edges-from ':mouse :minimum-width 200 :minimum-height 200
|     :blinker-p nil :expose-p t)
|   (:documentation :mixin
|     "Provides for a basic window to display the arrow graphic.
|     ARROW-PARAMETER-MIXIN is needed to position the figure within
|     the window. This flavor assumes window coordinates, with origin
|     at top left."))
```

2. Provide a `:show-lines` method to draw lines on the screen. We use essentially the same methods as in our original output module, but now we assume that the arguments are screen coordinates. We define separate `:compute-x` and `:compute-y` methods to transform the coordinates so that we can *shadow*

these methods when we define another flavor to handle LGP coordinates. To produce the lines we use the `:draw-line` method defined for `tv:graphics-mixin`, a component of `tv>window`. (In `:daemon` method combination, when two component flavors have primary methods for the same message, the method of the flavor listed earlier in the component ordering shadows, or replaces, the method of the flavor listed later. For more on method combination: See the section "Method Combination" in *Symbolics Common Lisp: Language Concepts*.)

```
|
|   ;;; Receives endpoint coordinates and draws lines on a window.
|   ;;; Arguments are alternating x- and y-coordinates of the end-
|   ;;; points of lines to be drawn. If there are more than two pairs
|   ;;; of coordinates, assumes that the endpoint of one line is the
|   ;;; starting point of the next. Sends messages for separate methods
|   ;;; to determine the actual coordinates. This is so that other
|   ;;; flavors can modify the coordinates. Draws a line by sending self
|   ;;; a :DRAW-LINE message, and so assumes that TV:GRAPHICS-MIXIN is
|   ;;; included somewhere to provide this method.
|   (defmethod (basic-arrow-window-mixin :show-lines)
|     (x y &rest x-y-pairs)
|     ;; First determine the starting point of the line. On
|     ;; subsequent trips through the loop, the last endpoint
|     ;; becomes the next starting point.
|     (loop for x0 = (send self ':compute-x x) then x1
|           for y0 = (send self ':compute-y y) then y1
|           ;; "Cddr" down the list created by making all but the
|           ;; first pair of coordinates an &rest argument
|           for (x1 y1) on x-y-pairs by #'cddr
|           ;; Determine the endpoint of the line
|           do (setq x1 (send self ':compute-x x1)
|                 y1 (send self ':compute-y y1))
|           ;; Draw the line
|           (send self ':draw-line
|                 x0 y0 x1 y1 tv:alu-iop t)))
|
|   ;;; Determines the x-coordinate of an endpoint of a line.
|   ;;; This is a separate method so that other flavors can shadow
|   ;;; it or add daemons to manipulate the coordinate.
|   (defmethod (basic-arrow-window-mixin :compute-x) (x)
|     (fixr x))
```

```

|   ;;; Determines the y-coordinate of an endpoint of a line.
|   ;;; Assumes that the argument already uses window coordinates,
|   ;;; with origin at top left. This is a separate method so that
|   ;;; other flavors can shadow it or add daemons to manipulate
|   ;;; the coordinate.
|   (defmethod (basic-arrow-window-mixin :compute-y) (y)
|     (fixr y))

```

3. Supply the **:compute-width-and-height** method required by **arrow-parameter-mixin**. We use the **:inside-size** message to **tv:sheet**, a component of **tv>window**. We use **zl:multiple-value** to set the instance variables **width** and **height**.

```

|   ;;; Finds the inside width and height of the window.
|   ;;; Sends self an :INSIDE-SIZE message, and so assumes that
|   ;;; TV:SHEET is included somewhere to provide this
|   ;;; method.
|   (defmethod (basic-arrow-window-mixin
|     :compute-width-and-height) ()
|     (multiple-value (width height)
|       (send self ':inside-size)))

```

4. Alter the computation of **top-y** to take account of the screen's origin at top left. We can do this in three ways:
- Define a new primary method for **:compute-top-y** to shadow the method we defined for **arrow-parameter-mixin**. We would have to be careful to place **basic-arrow-window-mixin** before **arrow-parameter-mixin** in the list of component flavors for any flavor we wanted to instantiate.
  - Define **:before** and **:after daemons** for **:compute-top-y**. The **:before** daemon would make **top-edge** negative and the **:after** daemon would make it positive again. (In **:daemon** method combination, **:before** methods for a message run before the primary method, and **:after** methods run after the primary method. If two component flavors have daemons for the same message, the **:before** method of the flavor listed earlier in the component ordering runs *before* the **:before** method of the flavor listed later, and the **:after** method of the flavor listed earlier runs *after* the **:after** method of the flavor listed later. For more on method combination: See the section "Method Combination" in *Symbolics Common Lisp: Language Concepts*.

- Define a *whopper* for **:compute-top-y**. This would do the same thing as the two daemons, except that when all the **:compute-top-y** methods were combined it would run outside any daemons. (A whopper wraps the execution of some code around the execution of a method, running before all **:before** daemons and after all **:after** daemons. For more on whoppers: See the special form **defwhopper** in *Symbolics Common Lisp: Language Dictionary*.

We define a new primary method in this case because it repeats relatively little code and makes the operation of the method clearer. If we used a whopper here, someone might mix in another flavor with daemons that would unexpectedly run inside our whopper.

```
|   ;;; Calculates y-coordinate of top right point of figure.
|   ;;; Finds vertical length of the figure by dividing the length
|   ;;; of top edge by 0.8. Centers the figure vertically within
|   ;;; the window. Gives the result in window coordinates, with
|   ;;; origin at top left. This method shadows that in
|   ;;; ARROW-PARAMETER-MIXIN.
|   (defmethod (basic-arrow-window-mixin :compute-top-y) ()
|     (setq top-y
|           (fixr (* 0.5 (- height (/ top-edge 0.8))))))
```

5. Calculate the figure's size and position and redisplay the window at appropriate times. We have to recompute the figure's size and position after the window is initialized and after its size or margins change. We have to redisplay the figure when the window is refreshed, but only if the window has no bit-save array or its size has changed. Before redisplaying, we have to clear the screen if the window *has* a bit-save array.

We perform these tasks by defining **:after** daemons for three messages that the system can send to a window: **:init**, **:change-of-size-or-margins**, and **:refresh**. You need daemons like these for most window-system applications.

```
|   ;;; Calculates size and position of figure after initialization.
|   (defmethod (basic-arrow-window-mixin :after :init) (ignore)
|     (send self ':compute-parameters))
```

```

|   ;;; Calculates size and position of figure after window change.
|   (defmethod (basic-arrow-window-mixin
|             :after :change-of-size-or-margins) (&rest ignore)
|     (send self ':compute-parameters))
|
|   ;;; Draws the figure when necessary after window is refreshed.
|   (defmethod (basic-arrow-window-mixin :after :refresh)
|     (&optional type)
|     ;; Draw figure if not restored from a bit-save array ...
|     (when (or (not tv:restored-bits-p)
|               ;; ... or size has changed.
|               (eq type ':size-changed))
|       ;; If restored from a bit-save array, clear screen first
|       (when tv:restored-bits-p
|         (send self ':clear-screen))
|       ;; Bind *DEST* to self
|       (let ((*dest* self))
|         ;; Draw the figure
|         (draw-arrow-graphic top-edge right-x top-y))))

```

We can now define a flavor of window, **basic-arrow-window**, built on our two mixin flavors and on **tv:window**. The order of combination of flavors is important. We need to include **basic-arrow-window-mixin** before **arrow-parameter-mixin** so that the **:compute-top-y** method for **basic-arrow-window-mixin** shadows that for **arrow-parameter-mixin**. We must also put **basic-arrow-window-mixin** before **tv:window** so that our **:after** daemons will run after any that **tv:window** or its components might provide.



```
| (defflavor basic-arrow-window ()  
|   (basic-arrow-window-mixin  
|     arrow-parameter-mixin  
|     tv:window)  
|   (:documentation :combination  
|     "Instantiable flavor providing a basic window for output.  
|     Though this flavor is instantiable, its methods assume that  
|     point coordinates use the window coordinate system, with  
|     origin at top left. To work with the current calculation  
|     module it needs another mixin to convert LGP to screen  
|     coordinates. In the component flavors, BASIC-ARROW-WINDOW-MIXIN  
|     must come before ARROW-PARAMETER-MIXIN and TV:WINDOW for  
|     shadowing and daemons to work correctly."))
```

We can actually make an instance of this flavor. We define no new methods for it, leaving all methods to component flavors. If we had a calculation module that used screen coordinates, **basic-arrow-window** would be the right flavor to use for screen output.

---

### 31.1.3 Converting LGP to Screen Coordinates

Because our calculation module uses LGP coordinates, we need another flavor of window to produce output. We define a flavor, **lgp-window-mixin**, to be mixed in with **basic-arrow-window**. We need a new instance variable, **scale-factor**, whose value is the ratio of LGP to screen pixel densities.

```

| (defflavor lgp-window-mixin
|   ((scale-factor 2.5))
|   ())
|   (:required-flavors basic-arrow-window)
|   (:documentation :mixin
|     "Converts LGP to screen coordinates and vice versa.
|     When mixed in with BASIC-ARROW-WINDOW, this flavor allows
|     window output with a calculation module that uses LGP
|     coordinates. The instance variable SCALE-FACTOR is the
|     ratio of LGP to screen pixel density. The methods take
|     the height and width of the window in screen pixels and
|     calculate the length of the top edge and the coordinates
|     of the top right point of the figure in LGP pixels. In
|     drawing lines on the window, the methods convert LGP to
|     window coordinates. These methods shadow those in
|     ARROW-PARAMETER-MIXIN and BASIC-ARROW-WINDOW-MIXIN.")

```

We next define new primary methods to incorporate the scale factor into the calculation of **top-edge**, **right-x**, and **top-y**. These methods shadow those defined for **arrow-parameter-mixin** and **basic-arrow-window-mixin**.

```

|   ;;; Calculates top edge in LGP pixels from screen proportions.
|   ;;; Multiplies length of smaller dimension, in screen pixels, by
|   ;;; proportion of this dimension to be filled by the figure.
|   ;;; Multiplies this by 0.8 to find top edge in screen pixels.
|   ;;; Corrects for higher density of LGP pixels. This method
|   ;;; shadows that of ARROW-PARAMETER-MIXIN.
|   (defmethod (lgp-window-mixin :compute-top-edge) ()
|     (setq top-edge
|       (fixr (* scale-factor 0.8 *fill-proportion*
|         (min width height))))

```

```

|   ;;; Calculates x-coord of top right point in LGP pixels.
|   ;;; Finds horizontal length of figure in screen pixels by
|   ;;; dividing top edge by 0.8. Centers figure horizontally
|   ;;; in window, correcting for higher density of LGP pixels.
|   ;;; This method shadows that of ARROW-PARAMETER-MIXIN.
| (defmethod (lgp-window-mixin :compute-right-x) ()
|   (setq right-x
|         (fixr (* 0.5 (+ (* width scale-factor)
|                         .(// top-edge 0.8))))))
|
|   ;;; Calculates y-coord of top right point in LGP pixels.
|   ;;; Finds vertical length of figure in screen pixels by
|   ;;; dividing top edge by 0.8. Centers figure vertically
|   ;;; in window, correcting for higher density of LGP pixels.
|   ;;; This method shadows those of ARROW-PARAMETER-MIXIN and
|   ;;; BASIC-ARROW-WINDOW-MIXIN.
| (defmethod (lgp-window-mixin :compute-top-y) ()
|   (setq top-y
|         (fixr (* 0.5 (+ (* height scale-factor)
|                         .(// top-edge 0.8))))))
|

```

Finally, we need to modify the coordinates used in the **:show-lines** method to take account of the scale factor and the difference in origins for LGP and screen coordinates. We define new methods for **:compute-x** and **:compute-y** to shadow the methods we defined for **basic-arrow-window-mixin**.

```

|   ;;; Converts x-coord of line endpoint from LGP to screen pixels.
|   ;;; Corrects for higher density of LGP pixels. This method shadows
|   ;;; that of BASIC-ARROW-WINDOW-MIXIN.
| (defmethod (lgp-window-mixin :compute-x) (x)
|   (fixr (// x scale-factor)))
|
|   ;;; Converts y-coord of line endpoint from LGP to screen pixels.
|   ;;; Corrects for higher density of LGP pixels and for screen origin
|   ;;; at top left. This method shadows that of BASIC-ARROW-WINDOW-MIXIN.
| (defmethod (lgp-window-mixin :compute-y) (y)
|   (fixr (- height (// y scale-factor))))
|

```

We can now define the flavor we will actually instantiate with **tv:make-window**. This flavor, **arrow-window**, is just a combination of **lgp-window-mixin** and **basic-arrow-window**.

```

| (defflavor arrow-window ()
|   (lgp-window-mixin basic-arrow-window)
|   (:documentation :combination
|     "Instantiable flavor for window output from LGP coordinates.
|     This flavor has all the features of BASIC-ARROW-WINDOW but
|     assumes that the calculation module uses LGP coordinates. This
|     is the flavor to instantiate for window output using the
|     current calculation module."))

```

---

#### 31.1.4 Flavors for LGP Output

We want to be able to direct output to an LGP or an LGP record file as well as to a window. We define another flavor, **lgp-pixel-mixin**, to be mixed in with **arrow-parameter-mixin**. We can set an instance variable to the output stream and make it *initable* so that we can specify the output stream when we make an instance of the flavor we build on **lgp-pixel-mixin**. The output stream will itself be an instance of a flavor.

```

| (defflavor lgp-pixel-mixin
|   (output-stream)
|   ()
|   :initable-instance-variables
|   (:required-flavors arrow-parameter-mixin)
|   (:documentation :mixin
|     "Provides methods for arrow graphic output on an LGP stream.
|     ARROW-PARAMETER-MIXIN is required to calculate the size of the
|     figure and position it in the center of the page. The method
|     assumes that coordinates are in LGP pixels. This flavor
|     should be mixed, along with ARROW-PARAMETER-MIXIN, into an
|     instantiable flavor for LGP output. When that flavor is
|     instantiated, the instance variable output-stream should be
|     initialized."))

```

The methods for this flavor need to do two things: determine the width and height of a page and handle **:show-lines** messages. We get the width and height from the values of instance variables for the flavor **lgp::basic-lgp-stream**. This flavor will be a component of the flavor we instantiate as the output stream.

```

|   ;;; Finds width and height of a page for LGP output.
|   ;;; This flavor is required by ARROW-PARAMETER-MIXIN. Finds the
|   ;;; values of two instance variables of LGP:BASIC-LGP-STREAM:
|   ;;; SI:PAGE-WIDTH and SI:PAGE-HEIGHT. Assumes that
|   ;;; LGP:BASIC-LGP-STREAM is included in output stream to provide
|   ;;; these instance variables.
|   (defmethod (lgp-pixel-mixin :compute-width-and-height) ()
|     (setq width (symeval-in-instance output-stream 'si:page-width)
|           height (symeval-in-instance output-stream 'si:page-height)))

```

The **:show-lines** method is similar to that for windows. Instead of using the **:draw-line** message to produce lines, we use two messages to **lgp::basic-lgp-stream**: **:send-command** and **:send-coordinates**.

```

|   ;;; Receives endpoint coordinates and draws lines on LGP stream.
|   ;;; Arguments are alternating x- and y-coordinates of endpoints of
|   ;;; lines to be drawn. If there are more than two pairs of
|   ;;; coordinates, assumes that the endpoint of one line is the
|   ;;; starting point of the next. Draws a line by sending output
|   ;;; stream :SEND-COMMAND messages for LGP commands and
|   ;;; :SEND-COORDINATE messages for LGP coordinates. Assumes that
|   ;;; flavor LGP:BASIC-LGP-STREAM is included in output stream to
|   ;;; provide these methods.
|   (defmethod (lgp-pixel-mixin :show-lines)
|     (x0 y0 &rest x-y-pairs)
|     ;; Send command and coordinates to start drawing lines
|     (send self ':send-command-and-coordinates #/m x0 y0)
|     ;; "Cddr" down the list created by making all but the first
|     ;; pair of coordinates an &rest argument
|     (loop for (x y) on x-y-pairs by #'cddr
|           ;; Send command and coordinates to draw a line
|           do (send self ':send-command-and-coordinates #/v x y)))

```

```

|   ;;; Sends line-drawing commands to LGP output stream.
|   ;;; :SEND-COMMAND transmits an LGP command. :SEND-COORDINATES
|   ;;; transmits coordinates of an endpoint of a line to be drawn.
|   ;;; Assumes that LGP:BASIC-LGP-STREAM is included in output stream
|   ;;; to provide these methods.
| (defmethod (lgp-pixel-mixin :send-command-and-coordinates) (cmd x y)
|   (send output-stream ':send-command cmd)
|   (send output-stream ':send-coordinates (fixr x) (fixr y)))

```

We can now define an instantiable flavor for the LGP stream that combines **lgp-pixel-mixin** and **arrow-parameter-mixin**.

```

| (def flavor lgp-pixel-stream ()
|   (lgp-pixel-mixin arrow-parameter-mixin)
|   (:documentation :combination
|     "Instantiable flavor for arrow output on LGP stream.
|     Assumes that the calculation module uses LGP coordinates.
|     When this flavor is instantiated, the LGP-PIXEL-MIXIN
|     instance variable OUTPUT-STREAM should be initialized.
|     The output stream can be directed to an LGP or a file,
|     but it must include flavor LGP:BASIC-LGP-STREAM for
|     output to work correctly."))

```

---

### 31.1.5 The Top-Level Function

We are ready to define a top-level function we can call to produce the graphic. We start by popping up a choose-variable-values window. We allow the user to specify screen, LGP, or file output. We also allow the user to choose values for the number of recursion levels and the proportion of the page or window to be filled. We let the user decide whether or not to stripe the arrows.

```

| (defvar *dest-string* "Screen"
|   "Destination of program output [Screen, LGP, or File]")
|
| (defvar *output-file* nil
|   "Pathname for LGP-record-file output")

```

```

|   ;;; Top-level function to call to produce arrow graphic.
|   ;;; Pops up a choose-variable-values window to let user specify
|   ;;; output destination, number of recursion levels, proportion
|   ;;; of smaller dimension of page or window to be filled, and
|   ;;; whether or not to stripe figure.
| (defun do-arrow ()
|   ;; Pop up a choose-variable-values window
|   (tv:choose-variable-values
|     '((*do-the-stripes* "Stripe the arrows?" :boolean)
|       (*max-depth* "Number of recursion levels" :number)
|       (*fill-proportion*
|         "Fraction of page or window to be filled" :number)
|       (*dest-string* "Output destination"
|         :choose ("Screen" "LGP" "File"))
|       (*output-file* "Pathname for file output" :PATHNAME))
|     ;; Make window wide enough to accommodate long pathnames
|     ;; and error messages
|     ':extra-width 20.
|     ;; Give user a chance to abort
|     ':margin-choices '("Do It" ("Abort" (signal 'sys:abort)))
|     ':label "Choose Options for Graphic"))

```

Next we need to take action depending on the output destination the user has chosen. If the variable *\*fill-proportion\** is zero, we just return *nil* no matter what the output destination. If the destination is "Screen", we make an instance of **arrow-window**. We use **tv:make-window**, which creates a new window each time we call **do-arrow**. We could also have defined a resource of arrow windows (using **defwindow-resource**), but we might want more than one selectable arrow window at a time.

If we have more than one arrow window, we want each to retain its own values for number of recursion levels, proportion of the window to be filled, and presence or absence of striping. We define three instance variables for **basic-arrow-window-mixin** and make them *initable*. We initialize them when we call **tv:make-window** from **do-arrow**. We change the **:after** daemons for **basic-arrow-window-mixin** to bind the special variables to the instance-variable values.

```

(defflavor basic-arrow-window-mixin
|   (do-stripes max-dep fill-prop)
|   ())
|   :initable-instance-variables
|   (:required-flavors arrow-parameter-mixin tv:window)
|   (:default-init-plist
|     :edges-from ':mouse :minimum-width 200 :minimum-height 200
|     :blinker-p nil :expose-p t)
|   (:documentation :mixin ...))

(defmethod (basic-arrow-window-mixin :after :init) (ignore)
|   (let ((*fill-proportion* fill-prop))
|     (send self ':compute-parameters)))

(defmethod (basic-arrow-window-mixin
|   :after :change-of-size-or-margins) (&rest ignore)
|   (let ((*fill-proportion* fill-prop))
|     (send self ':compute-parameters)))

(defmethod (basic-arrow-window-mixin :after :refresh)
|   (&optional type)
|   ;; Draw figure if not restored from a bit-save array ...
|   (when (or (not tv:restored-bits-p)
|             ;; ... or size has changed.
|             (eq type ':size-changed))
|     ;; If restored from a bit-save array, clear screen first
|     (when tv:restored-bits-p
|       (send self ':clear-screen))
|     ;; Bind global variables to self and instance variables
|     (let ((*dest* self)
|           (*do-the-stripes* do-stripes)
|           (*max-depth* max-dep))
|       ;; Draw the figure
|       (draw-arrow-graphic top-edge right-x top-y))))

```







---

### 31.1.6 The Arrow Window: Interaction, Processes, and the Mouse

Suppose we want to let the user modify the characteristics of the graphic for each window. The user might want to change the presence or absence of striping, the number of recursion levels, or the proportion of the window to be filled.

One way to install this option is to associate each window with its own process and let the process run in a loop. If the user clicks right on the window, we pop up a choose-variable-values window. When the user is finished, we refresh the window and wait for the next mouse click.

We can associate a window with a process by including the flavor `tv:process-mixin` in `basic-arrow-window`. When we make the window (using `tv:make-window`), we specify a `:process` init option whose argument is the name of the top-level function for the process. When the window is created, a new process is created as well. When the window is exposed, the process's top-level function is called with one argument, the window.

```
(defflavor basic-arrow-window ()
  (basic-arrow-window-mixin
   arrow-parameter-mixin
   tv:process-mixin
   tv>window)
  (:documentation :combination ...))
```



```

| (defmethod (basic-arrow-window-mixin
|       :who-line-documentation-string) ()
|   "Provides a mouse documentation line for the window.
|   The only option is to click right and pop up a
|   choose-variable-values window of options for changing
|   the graphic on this window."
|   "R: Choose-variable-values options for changing figure on this window")

```

We can now write the process function **window-loop**. This function just sends a **:main-loop** message to the window. We define **:main-loop** as a method of **basic-arrow-window-mixin**. The method consists of an **error-restart-loop** so that we can return to top level if **sys:abort** or an error is signalled. We send the window an **:any-tyi** message. If the user clicks right, we pop up a choose-variable-values window with the window's current value of the variables. When the user exits, we refresh the window and wait for another click. If the user aborts, **sys:abort** is signalled, and we restart the loop.

```

|   ;;; Top-level function for process associated with arrow window.
|   ;;; The function is called when the window is created. Argument is
|   ;;; the window. The function sends the window a :MAIN-LOOP message.
|   ;;; This method should be the actual command loop for the process.
| (defun window-loop (window)
|   (send window ':main-loop))

```

```

|   ;; Command loop for window associated with a separate process.
|   ;; Consists of an error-restart-loop that handles restarts from errors
|   ;; and sys:abort.  Waits for mouse input.  If a right click, pops up a
|   ;; choose-variable-values window to change characteristics of the
|   ;; figure.  On exit, sets instance variables to the new values and
|   ;; refreshes the window, then waits for another mouse click.  Assumes
|   ;; blips are lists of the form provided by TV:LIST-MOUSE-BUTTONS-MIXIN.
| (defmethod (basic-arrow-window-mixin :main-loop) ()
|   ;; Run forever in a loop.  Offer a restart handler if an error
|   ;; or SYS:ABORT is signalled.
|   (error-restart-loop ((error sys:abort) "Arrow Window Top Level")
|     ;; Wait for input
|     (let ((char (send self ':any-tyi)))
|       ;; Pop up window if input is a list ...
|       (when (and (listp char)
|                 ;; ... and a mouse click ...
|                 (eq (first char) ':mouse-button)
|                 ;; ... and a single click on the right button.
|                 (eq (second char) #\mouse-r-1))
|         ;; Bind global variables to instance-variable values
|         (let ((*do-the-stripes* do-stripes)
|               (*max-depth* max-dep)
|               (*fill-proportion* fill-prop))
|           ;; Pop up a choose-variable-values window
|           (tv:choose-variable-values
|             '((*do-the-stripes* "Stripe the arrows?" :boolean)
|               (*max-depth* "Number of recursion levels" :number)
|               (*fill-proportion*
|                 "Fraction of window to be filled" :number))
|             ;; Make the window wide to provide enough room for error
|             ;; messages.
|             ':extra-width 20
|             ;; Give the user a chance to abort
|             ':margin-choices '("Do It" ("Abort" (signal 'sys:abort)))
|             ':label "Choose Options For Graphic")
|           ;; Set instance variables to the new values
|           (setq do-stripes *do-the-stripes*
|                 max-dep *max-depth*
|                 fill-prop *fill-proportion*)
|           ;; Recompute size and position of the figure
|           (send self ':compute-parameters)
|           ;; Send :REFRESH message with argument of ':new-vals to make

```

```
|           ;; sure the figure is redrawn if there is a bit-save array
|           (send self ':refresh ':new-vals))))))
```

We need to change the `:after :refresh` method of `basic-arrow-window-mixin` so that it redraws the figure when the values are changed even if the window has a bit-save array.

```
(defmethod (basic-arrow-window-mixin :after :refresh)
  (&optional type)
  ;; Draw figure if not restored from a bit-save array ...
  (when (or (not tv:restored-bits-p)
            ;; ... or size has changed ...
            (eq type ':size-changed)
            |           ;; ... or new values for figure parameters.
            |           (eq type ':new-vals))
        ;; If restored from a bit-save array, clear screen first
        (when tv:restored-bits-p
          (send self ':clear-screen))
        ;; Bind global variables to self and instance variables
        (let ((*dest* self)
              (*do-the-stripes* do-stripes)
              (*max-depth* max-dep))
          ;; Draw the figure
          (draw-arrow-graphic top-edge right-x top-y))))
```

Note that we can also manipulate the windows we create by using the [Split Screen] and [Edit Screen] options from the System menu. We might have more than one arrow window on the screen at the same time. We might redisplay the figures on these windows at the same time. In this case, the scheduler might switch between the arrow window processes, allowing each to run for a time until all redispays are complete.

Remember that we took care to *bind* rather than *set* the global variables in the calculation module that hold the state of each arrow. We want the values of some variables to be different in each window. Each process maintains its own bindings for variables. When the scheduler switches processes, bindings in the old process are undone and saved. They are restored when the old process resumes. But if we had set the variables, the program would not have run correctly when the scheduler switched processes. The new process might have used variable values set in the old process.

### 31.1.7 Defining Flavors to Signal Conditions

We want to add one more refinement to the output module. In our choose-variable-values windows, the variable type keywords, such as **:number** and **:pathname**, provide for some error checking when users choose new values. But two of our numeric variables have further restrictions: **\*max-depth\*** must be a nonnegative integer, and **\*fill-proportion\*** must be a fraction between 0 and 1.

The function **tv:choose-variable-values** has a **:function** option that lets us name a function to be called whenever an item is to be changed. We can use this function to check the values of our two variables and signal a condition if the values are bad. We then print a message on the window and ask the user to proceed by supplying a new value.

We start by defining flavors for the conditions we signal. We define a general class of error conditions called **bad-arrow-variable**. We then define two flavors built on **bad-arrow-variable**: **bad-arrow-depth** for improper values of **\*max-depth\*** and **bad-arrow-fill-proportion** for improper values of **\*fill-proportion\***. For each of these instantiable flavors we define a **:report** method and a **:proceed** method. The **:report** method prints a string identifying the condition. The **:proceed** method allows the user to proceed from the condition, in this case by supplying a new value. We could have more than one **:proceed** method if we had other ways of proceeding. **:proceed** methods are combined using **:case** method combination.

If we want to create conditions for bad values of other variables in the future, we can simply define new flavors built on **bad-arrow-variable**.

```
| (def flavor bad-arrow-variable () (error)
|   (:documentation
|     "Noninstantiable class of bad-variable conditions.
|     The user might set some variables to impermissible values.
|     These conditions are to permit checking for bad values
|     beyond the system's error checking. Instantiable condition
|     flavors for specific variables should be built on this
|     flavor."))
```



```
| (defflavor bad-arrow-depth () (bad-arrow-variable)
| (:documentation
| "Proceedable condition: bad value for *MAX-DEPTH*.
| An instantiable condition flavor for impermissible values
| of *MAX-DEPTH*, the number of recursion levels in the
| figure.")
|
| ;;; Prints string on stream to report bad *MAX-DEPTH* value
| (defmethod (bad-arrow-depth :report) (stream)
| (format stream "No. of levels was not a ~
| nonnegative fixnum."))
|
| ;;; Proceed type method for supplying new value of *MAX-DEPTH*
| (defmethod (bad-arrow-depth :case :proceed :new-depth)
| (&optional (dep (prompt-and-read
| ':number
| "Supply new value for ~
| no. of recursion levels: ")))
| "Supply a new value for number of recursion levels."
| (values ':new-depth dep))
|
| (defflavor bad-arrow-fill-proportion () (bad-arrow-variable)
| (:documentation
| "Proceedable condition: bad value for *FILL-PROPORTION*.
| An instantiable condition flavor for impermissible values of
| *FILL-PROPORTION*, the fraction of the smaller dimension of
| the page or window that the figure is to fill.")
|
| ;;; Prints string on stream to report bad *FILL-PROPORTION* value
| (defmethod (bad-arrow-fill-proportion :report) (stream)
| (format stream "Proportion was not a fraction between ~
| 0 and 1."))
```

```
|   ;; Proceed type method for new value of *FILL-PROPORTION*  
|   (defmethod (bad-arrow-fill-proportion :case :proceed  
|           :new-proportion)  
|       (&optional (prop (prompt-and-read  
|           ':number  
|           "Supply new fraction of bounds ~  
|           be filled: ")))  
|       "Supply a new fraction of page or window to be filled."  
|       (values ':new-proportion prop))
```

Next we write the function, **check-item**, to be called when a variable value is changed. The function is called with four arguments: the choose-variable-values window, the variable, and the variable's old and new values. We use **condition-bind** to bind a handler for our two conditions. This handler will be called if we signal the conditions from within the **condition-bind**. If we do find a bad variable value, we expect the call to **signal** to return the two values from the **:proceed** method: the proceed type and the new variable value. We then check the new value and, if it is good, set the variable to the new value. Finally, we refresh the window and return **t**.

```

|   ;;; Called when a value changes in choose-variable-values window.
|   ;;; Arguments are the window, the variable, and its old and new values.
|   ;;; Binds handlers for conditions for impermissible values.  If new
|   ;;; value is OK, sets variable to the new value, refreshes window, and
|   ;;; returns t.  If value is not OK, signals the appropriate condition.
|   ;;; When SIGNAL returns, presumably with a new variable value, checks
|   ;;; the new value in the same way it checks a new value that comes
|   ;;; from the window.
|   (defun check-item (cvv-window var old-val new-val)
|     ;; We don't use the old value.  To avoid a compiler complaint,
|     ;; just evaluate it and ignore it.  We could also use IGNORE
|     ;; instead of OLD-VAL in the arglist, but then the arglist
|     ;; would be less meaningful.
|     old-val
|     ;; Bind handlers for the conditions we might signal
|     (condition-bind ((bad-arrow-depth 'bad-arrow-var-handler)
|                     (bad-arrow-fill-proportion
|                     'bad-arrow-var-handler))
|     (when (eq var '*max-depth*)
|       ;; *MAX-DEPTH* must be nonnegative fixnum
|       (loop until (and (fixp new-val) (≥ new-val 0))
|         ;; If it's not, bind QUERY-IO to the window and
|         ;; signal a condition.  SIGNAL should return
|         ;; two values, the proceed type and the new
|         ;; value from the proceed method.  Ignore the
|         ;; proceed type and set NEW-VAL to the new
|         ;; value.
|         do (let ((query-io cvv-window))
|             (multiple-value (nil new-val)
|                             (signal 'bad-arrow-depth))))))
|     (when (eq var '*fill-proportion*)
|       ;; *FILL-PROPORTION* must be between 0 and 1
|       (loop until (and (≥ new-val 0) (≤ new-val 1))
|         ;; If it's not, bind QUERY-IO to the window and
|         ;; signal a condition.  SIGNAL should return
|         ;; two values, the proceed type and the new
|         ;; value from the proceed method.  Ignore the
|         ;; proceed type and set NEW-VAL to the new
|         ;; value.
|         do (let ((query-io cvv-window))
|             (multiple-value (nil new-val)
|                             (signal 'bad-arrow-fill-proportion))))))

```

```

|         ;; Variable value is now OK.  Set variable to the new value.
|         ;; Note that we DO want to evaluate VAR.
|         (set var new-val)
|         ;; Refresh the window
|         (send cvv-window ':refresh)
|         ;; Return t
|         t))

```

Next we need to add the **:function** option to our calls to **tv:choose-variable-values** in the function **do-arrows** and the **:main-loop** method of **basic-arrow-window-mixin**:

```

(defun do-arrow ()
  ;; Pop up a choose-variable-values window
  (tv:choose-variable-values
   '((*do-the-stripes* "Stripe the arrows?" :boolean)
     (*max-depth* "Number of recursion levels" :number)
     (*fill-proportion*
      "Fraction of page or window to be filled" :number)
     (*dest-string* "Output destination"
      :choose ("Screen" "LGP" "File"))
     (*output-file* "Pathname for file output" :pathname))
   ;; Make window wide enough to accommodate long pathnames
   ;; and error messages
   ':extra-width 20.
  |   ;; Call this function when a value is changed
  |   ':function 'check-item
  |   ;; Give user a chance to abort
  |   ':margin-choices '("Do It" ("Abort" (signal 'sys:abort)))
  |   ':label "Choose Options for Graphic")
  |   .
  |   .

```

```

(defmethod (basic-arrow-window-mixin :main-loop) ()
  ;; Run forever in a loop. Offer a restart handler if an error
  ;; or sys:abort is signalled.
  (error-restart-loop ((error sys:abort) "Arrow Window Top Level")
    ;; Wait for input
    (let ((char (send self ':any-tyi)))
      ;; Pop up window if input is a list ...
      (when (and (listp char)
                 ;; ... and a mouse click ...
                 (eq (first char) ':mouse-button)
                 ;; ... and a single click on the right button.
                 (eq (second char) #\mouse-r-1))
        ;; Bind global variables to instance-variable values
        (let ((*do-the-stripes* do-stripes)
              (*max-depth* max-dep)
              (*fill-proportion* fill-prop))
          ;; Pop up a choose-variable-values window
          (tv:choose-variable-values
            '((*do-the-stripes* "Stripe the arrows?" :boolean)
              (*max-depth* "Number of recursion levels" :number)
              (*fill-proportion*
                "Fraction of window to be filled" :number))
            ;; Make the window wide to provide enough room for error
            ;; messages.
            ':extra-width 20
            | ;; Call a function to check for errors when values change
            | ':function 'check-item
            ;; Give the user a chance to abort
            ':margin-choices '("Do It" ("Abort" (signal 'sys:abort)))
            ':label "Choose Options for Graphic")
            ;; Set instance variables to the new values
            (setq do-stripes *do-the-stripes*
                  max-dep *max-depth*
                  fill-prop *fill-proportion*)
            ;; Recompute size and position of the figure
            (send self ':compute-parameters)
            ;; Send :REFRESH message with argument of ':new-vals to make
            ;; sure the figure is redrawn if there is a bit-save array
            (send self ':refresh ':new-vals))))))

```

Finally, we need to write a handler for the two conditions. When a condition is signalled, the handler is called with one argument,

the object of the flavor of condition that is signalled. In **check-item**, we call **signal** with **zl:query-io** bound to the choose-variable-values window. The handler checks to be sure there is a **proceed** type for the object. If so, the handler turns on a blinker on the window and sends the **:report** and **:proceed** messages to the condition object. Finally, it turns off the blinker and passes back to its caller the two values that the **:proceed** method returns.

Actually, the handler we define doesn't depend on the binding of **zl:query-io** to the window. If **zl:query-io** is not bound to a window – that is, to an instance of a flavor built on **tv:sheet** – the handler won't try to turn on a blinker. If **zl:query-io** is bound to a window, the handler first looks (using **tv:sheet-following-blinker**) for an existing blinker that follows the cursor. If it doesn't find one, it makes a new blinker (using **tv:make-blinker**). It encloses the handling operation in an **unwind-protect** to be sure that the blinker is turned off in case of a nonlocal exit.

```

|   ;;; Handler for bad value of *MAX-DEPTH* or *FILL-PROPORTION*.
|   ;;; Argument is the condition object created by SIGNAL.  Uses QUERY-IO
|   ;;; stream to report condition.  Sends the condition object a :PROCEED
|   ;;; message and passes back the values it returns.
|   (defun bad-arrow-var-handler (cond-obj &aux bl)
|     ;; Find out whether this object has the right proceed type.
|     ;; If not, return nil.
|     (if (send cond-obj ':proceed-type-p
|               (cond ((typep cond-obj 'bad-arrow-depth) ':new-depth)
|                     ((typep cond-obj 'bad-arrow-fill-proportion)
|                      ':new-proportion)))
|         ;; Enclose the handling operation in an UNWIND-PROTECT so that
|         ;; if we use a blinker we are sure to turn it off
|         (unwind-protect
|           (progn
|             ;; Use a blinker if the QUERY-IO stream is a window
|             (setq bl (if (typep query-io 'tv:sheet)
|                          ;; If a cursor-following blinker exists, use it
|                          (or (tv:sheet-following-blinker query-io)
|                              ;; Otherwise, make a new blinker
|                              (tv:make-blinker query-io
|  'tv:rectangular-blinker
|  ':follow-p t))))
|             ;; If a blinker, make it blink
|             (if bl (send bl ':set-visibility ':blink))
|             ;; Alert the user
|             (tv:beep)
|             ;; Send a report, presumably describing the condition
|             (send cond-obj ':report query-io)
|             ;; Send object a :PROCEED message and return the values
|             ;; that the method returns
|             (send cond-obj ':proceed
|                   (cond ((typep cond-obj 'bad-arrow-depth) ':new-depth)
|                         ((typep cond-obj 'bad-arrow-fill-proportion)
|                          ':new-proportion))))
|             ;; If a blinker, turn it off
|             (if bl (send bl ':set-visibility nil))))))

```

After we have defined all the flavors and methods for the output module, we insert a **compile-flavor-methods** form in the file. Without this macro, combined methods are compiled and flavor data structures generated when we make the first instance of a

flavor – that is, at run time. **compile-flavor-methods** speeds run-time operation by causing combined methods to be compiled at compile time and data structures to be generated at load time. It is useful only for flavors that will be instantiated, not for flavors that are only components of instantiated flavors.

```
| (compile-flavor-methods arrow-window lgp-pixel-stream
|                               bad-arrow-depth bad-arrow-fill-proportion)
```

## 31.2 Programming Aids for Flavors and Windows

Some editor commands and Lisp functions provide information about flavors. You can find out about component flavors, methods, instance variables, init keywords, and documentation. Using the Inspector, you can examine instance variables and methods for instances of flavors: See the section "Using the Inspector", page 335. If a flavor has gettable instance variables, you can obtain their values by sending messages to instances of the flavor.

These commands and functions are useful for finding information about windows as well. Because windows are instances of flavors, you can retrieve characteristics that are stored in gettable instance variables by sending messages to the windows. See the section "Using the Window System" in *Programming the User Interface, Volume B*. If a window is exposed, you can examine and alter some characteristics by clicking on the [Attributes] item in the System menu. Clicking on [Attributes] pops up a choose-variable-values window for such characteristics as font, label, margins, and vertical spacing between lines.

As with other definitions, Edit Definition (M-. ) prepares to edit definitions of flavors and methods. For a description of how to use this command to edit method definitions: See the section "Methods", page 378.

---

### 31.2.1 General Information on Flavors

The facilities that display general information about a flavor are Describe Flavor (M-X) and **describe-flavor**. These display somewhat different descriptions of a flavor.

A useful predicate for instances of flavors is **zl:typep**. Given an instance and a flavor name, **zl:typep** returns **t** if the instance includes the flavor as a component.

---

#### Example

In handling bad values for the variables **\*max-depth\*** and **\*fill-proportion\***, we want to be sure that **zl:query-io** is bound to a window before turning on a blinker. We find out whether the object bound to **zl:query-io** is built on **tv:sheet** by using **zl:typep**:



(typep query-io 'tv:sheet)

---

### Reference

Describe Flavor (m-X)

Displays a description of a flavor that includes the names of instance variables and component flavors and any documentation added by the **:documentation** option for **defflavor**. Also displays init keywords and inherited methods and instance variables. Names of flavors and methods in the display are mouse sensitive.

(describe-flavor *flavor-name*)

Prints a description of a flavor that includes the names of instance variables and component flavors and any documentation added by the **:documentation** option for **defflavor**.

(typep *arg* *type*)

When *arg* is an instance of a flavor and *type* is a flavor name, returns **t** if the instance includes the flavor as a component or **nil** if it does not. If *type* is omitted, returns a symbol representing the flavor of the instance.

---

### 31.2.2 Methods

Four Zmacs commands display information about the methods that handle messages to instances of flavors. For instances of flavors built on **si:vanilla-flavor** – that is, for nearly all flavors – you can send messages to find out which messages the object handles and whether or not it handles a specific message.

You can use the Zmacs command Edit Definition (m-.) to edit the definition of a method. Specify a method by typing a representation of its function spec. This is a list of the following form:

**(:method *flavor type message*)**

When typing this representation for Edit Definition (`m-.`), *type* is optional. If the method has a type, Zmacs will try to find the definition and ask you whether or not that definition is the one you want.

You might know the name of a method but not the name of its flavor. Use List Methods (`m-X`) to find methods for all flavors that handle a message. You can click on one of the method names displayed to edit its definition.

---

### Example

We want to edit the definition of the `:main-loop` method of `basic-arrow-window-mixin`. We use Edit Definition (`m-.`) and type:

```
(:method basic-arrow-window-mixin :main-loop)
```

---

### Example

We want to find out which methods handle `:show-lines` messages and how the methods handle the messages. List Methods (`m-X`) displays the following methods:

```
Methods for :SHOW-LINES
(:METHOD BASIC-ARROW-WINDOW-MIXIN :SHOW-LINES)
(:METHOD LGP-PIXEL-MIXIN :SHOW-LINES)
```

We can click on one of the method names or press `c-.` to edit the definition. We also could have found the source code directly by using Edit Methods (`m-X`).

---

### Example

We want to find out which methods are called when the system sends an `:init` message to `arrow-window`. List Combined Methods (`m-X`) prompts for message and flavor names and displays the following methods, in the order in which they are called:

```

Combined method for :INIT message to ARROW-WINDOW flavor
(:METHOD TV:SHEET :WRAPPER :INIT)
(:METHOD TV:STREAM-MIXIN :BEFORE :INIT)
(:METHOD TV:BORDERS-MIXIN :BEFORE :INIT)
(:METHOD TV:ESSENTIAL-LABEL-MIXIN :BEFORE :INIT)
(:METHOD TV:ESSENTIAL-WINDOW :BEFORE :INIT)
(:METHOD TV:SHEET :INIT)
(:METHOD TV:ESSENTIAL-SET-EDGES :AFTER :INIT)
(:METHOD TV:LABEL-MIXIN :AFTER :INIT)
(:METHOD TV:PROCESS-MIXIN :AFTER :INIT)
(:METHOD BASIC-ARROW-WINDOW-MIXIN :AFTER :INIT)

```

---

### Reference

- |                                                 |                                                                                                                                                                               |
|-------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| List Methods (m-X)                              | Lists methods for all flavors that handle a specified message. Press c-. to edit the definitions of the methods listed.                                                       |
| Edit Methods (m-X)                              | Prepares to edit definitions of methods for all flavors that handle a specified message. Press c-. to edit subsequent definitions.                                            |
| List Combined Methods (m-X)                     | Lists all the methods that would be called if a specified message were sent to an instance of a specified flavor. Press c-. to edit the definitions of the methods listed.    |
| Edit Combined Methods (m-X)                     | Prepares to edit definitions of methods that would be called if a specified message were sent to an instance of a specified flavor. Press c-. to edit subsequent definitions. |
| <b>(send <i>instance</i> 'which-operations)</b> | Returns a list of messages that <i>instance</i> can handle.                                                                                                                   |

(send *instance* **:operation-handled-p** *message*)

Returns **t** if *instance* has a handler for *message* or **nil** if it does not.

(get-handler-for *object* *message*) Returns the method that handles *message* to *object*, or **nil** if *object* has no handler for *message*.

---

### 31.2.3 Init Keywords

**si:flavor-allowed-init-keywords** retrieves the init keywords allowed for a flavor.

---

#### Example

We want to find the allowed init keywords for **lgp-pixel-stream**. **si:flavor-allowed-init-keywords** returns the following list:

```
(:DO-STRIPES :FILL-PROP :MAX-DEP :OUTPUT-STREAM)
```

These are all keywords for initable instance variables, the first three from **arrow-parameter-mixin** and the last from **lgp-pixel-mixin**.

---

#### Reference

(**si:flavor-allowed-init-keywords** *flavor-name*)

Returns a list of any init keywords a flavor can take.



## 32. Calculation Module for the Sample Program

The program used as an example in this document draws the recursive arrow graphic on the document's cover. This section contains Lisp code that calculates coordinates for the endpoints of the lines that compose the figure. The code produces output by sending messages to instances of flavors defined in another file. For the code for the flavors and methods that mediate between the program and the system output operations: See the section "Output Module for the Sample Program", page 403. For a reproduction of the LGP graphic the program produces: See the section "Graphic Output of the Sample Program", page 425.

```
;;; -*- Mode: LISP; Package: (GRAPHICS GLOBAL 1000); Base: 10 -*-  
;;; Copyright (c) 1983 Symbolics, Inc.
```

```
##|
```

This file contains the calculation module for a program that reproduces the recursive arrow graphic printed on the covers of most Symbolics documents. The module calculates the coordinates of the endpoints of line segments to be drawn. It transmits these coordinates to a separate output module, which contains the code needed to produce the figure on an appropriate output device.

We use paper coordinates, origin at bottom left.

Each arrow in the figure can be seen as inscribed in a square whose apex is at (apex-x, apex-y). Each arrow has a head and a shaft. Top-edge is the top edge of each arrow, one of the sides of the arrowhead. There are two classes of arrow in the figure: The small arrows are the general case, and the large, outer arrow is unique. The differences are the structures of the shafts and the recursive appearance of the small arrows.

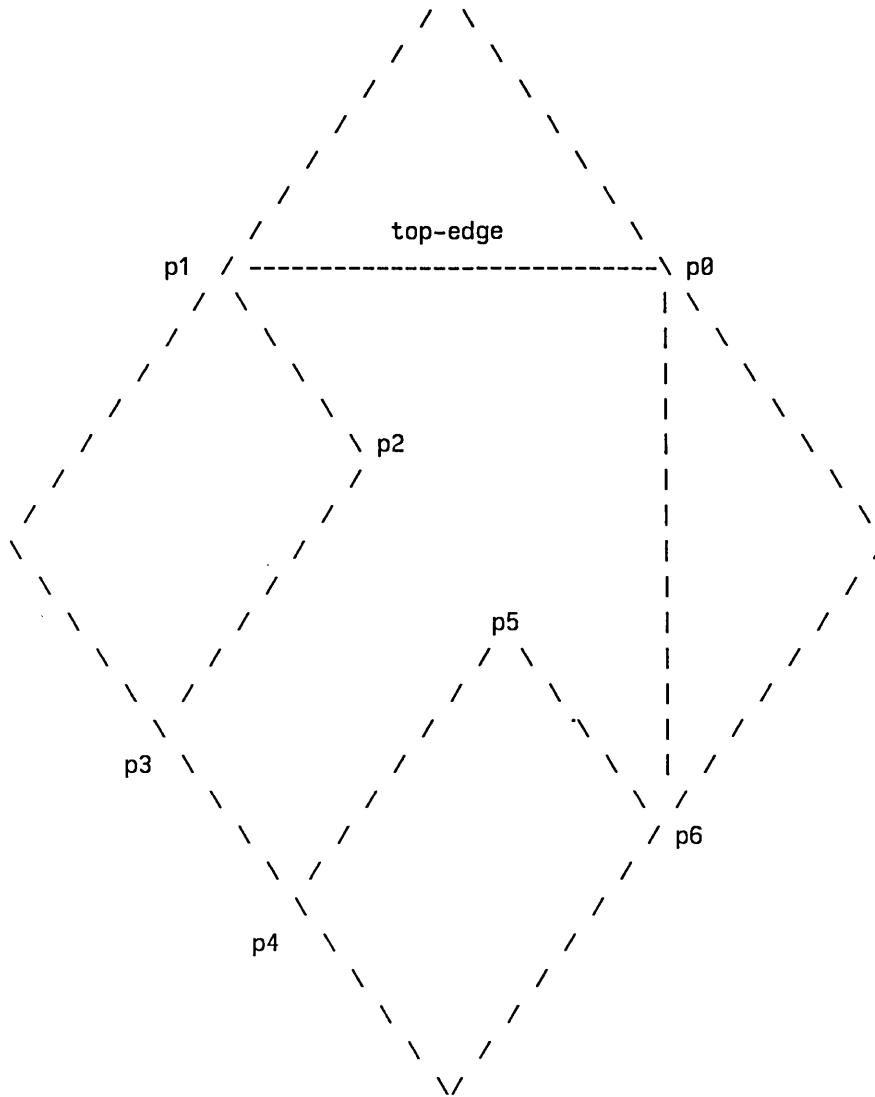
The module uses special variables to store information about the current arrow, including the length of the top edge and the coordinates of the vertexes.

The module first calculates coordinates for the vertexes of the large, outer arrow. If the arrows are to be striped, it determines the endpoints of the lines that make up the large arrow's stripes, first in the head and then in the shaft.

The module then recursively calculates coordinates for each of the small arrows inside the figure. It outlines and stripes one arrow at a time. For each arrow, the module first calculates the coordinates of the vertexes of the head. If the arrows are to be striped, it then determines the coordinates of the endpoints of the lines that make up the current arrow's stripes, first in the head and then in the shaft.

The output module initiates the calculation module by calling DRAW-ARROW-GRAPHIC with three arguments: the length of the figure's top edge and the coordinates of the top right point (p0 in the large arrow). This module transmits coordinates to the output module by sending :SHOW-LINES messages to instances of output flavors. The arguments to :SHOW-LINES are the coordinates of the endpoints of lines to be drawn. The current instance of the output flavor is the value of the special variable \*DEST\*.

(apex-x, apex-y)



Points 3 and 4 are obscured, except in the case of the big arrow.

```
||#
```

```
;;; Following are declarations for special variables and constants
```

```
(defconst *d1* 0.15
```

```
  "Proportion of distance filled in between upper right stripes")
```



```
(defconst *d2* 0.75
  "Proportion of distance filled in between lower left stripes")
```

```
(defconst *stripe-distance* 20
  "Horizontal distance in pixels between stripes of large arrow")
```

```
(defconst *max-depth* 7
  "Number of levels of recursion")
```

```
(defconst *do-the-stripes* t
  "If T, permits striping")
```

```
(defconst *dest* nil
  "Object to which output is sent")
```

```
(defvar *depth* 0
  "Current level of recursion")
```

```
(defvar *top-edge* nil
  "Length of the top edge of the arrow")
```

```
(defvar *top-edge-2* nil
  "Half the length of the top edge of the arrow")
```

```
(defvar *top-edge-4* nil
  "One-fourth the length of the top edge of the arrow")
```

```
(defvar *x2* nil
  "X-coord of projection of lower left stripe on top edge")
```

```
(defvar *stripe-d* nil
  "Horizontal distance in pixels between stripes")
```

```
(defvar *p0x* nil
  "X-coordinate of the tip of the arrow")
```

```
(defvar *p0y* nil
  "Y-coordinate of the tip of the arrow")
```

```
(defvar *p1x* nil
  "X-coordinate of point p1 in the arrow")
```

```
(defvar *p1y* nil
  "Y-coordinate of point p1 in the arrow")

(defvar *p2x* nil
  "X-coordinate of point p2 in the arrow")

(defvar *p2y* nil
  "Y-coordinate of point p2 in the arrow")

(defvar *p3x* nil
  "X-coordinate of point p3 in the arrow")

(defvar *p3y* nil
  "Y-coordinate of point p3 in the arrow")

(defvar *p4x* nil
  "X-coordinate of point p4 in the arrow")

(defvar *p4y* nil
  "Y-coordinate of point p4 in the arrow")

(defvar *p5x* nil
  "X-coordinate of point p5 in the arrow")

(defvar *p5y* nil
  "Y-coordinate of point p5 in the arrow")

(defvar *p6x* nil
  "X-coordinate of point p6 in the arrow")

(defvar *p6y* nil
  "Y-coordinate of point p6 in the arrow")
```

```
;;; Following are the controlling functions for this module
```

```

;;; Function controlling the calculation module.
;;; Controls the calculation of the coordinates of the endpoints of the
;;; lines that make up the figure. The three arguments are the length of
;;; the top edge and the coordinates of the top right point of the large
;;; arrow. DRAW-ARROW-GRAPHIC calls DRAW-BIG-ARROW to draw the large arrow
;;; and then calls DO-ARROWS to draw the smaller ones.
(defun draw-arrow-graphic (*top-edge* *p0x* *p0y*)
  ;; Bind global variables
  (let ((*top-edge-2* (/ *top-edge* 2))
        (*top-edge-4* (/ *top-edge* 4))
        ;; Compute horizontal distance between stripes in the large
        ;; arrow, assuming 64 stripes in the large arrowhead.
        (*stripe-distance* (/ *top-edge* 64)))
    (draw-big-arrow) ;Draw large arrow
    ;; Length of the top-edge for the first small arrow is half the
    ;; length for the large arrow. Bind new coordinates for the top
    ;; right point of the small arrow.
    (let ((*top-edge* *top-edge-2*)
          (*p0x* (- *p0x* *top-edge-2*))
          (*p0y* (- *p0y* *top-edge-2*))
          (*depth* 0))
      (do-arrows))) ;Draw small arrows

```

```

;;; Recursive function controlling drawing of the small arrows.
;;; If below the maximum recursion level, draws a small arrow. Binds
;;; new values for depth, top edge, and coordinates of top right point,
;;; and calls self recursively to draw a left-hand child arrow. Binds
;;; special variables again and calls self to draw a right-hand child
;;; arrow.
(defun do-arrows ()
  ;; Don't exceed maximum recursion level
  (when (< *depth* *max-depth*)
    ;; Bind values for half and one-fourth of top edge
    (let ((*top-edge-2* (/ *top-edge* 2))
          (*top-edge-4* (/ *top-edge* 4)))
      (draw-arrow) ;Draw a small arrow
      ;; Increment depth. Divide top edge in half. Bind new
      ;; coordinates for top right point of next arrow.
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (+ *top-edge-4* (- *p0x* *top-edge*)))
            (*p0y* (- *p0y* *top-edge-4*)))
        ;; Draw a left-hand child arrow
        (do-arrows))
      ;; Increment depth. Divide top edge in half. Bind new
      ;; coordinates for top right point of next arrow.
      (let ((*depth* (1+ *depth*))
            (*top-edge* *top-edge-2*)
            (*p0x* (- *p0x* *top-edge-4*)
            (*p0y* (+ *top-edge-4* (- *p0y* *top-edge*))))
        ;; Draw a right-hand child arrow
        (do-arrows))))))

;;; The following functions are common to the large and small arrows

```

```

;;; Calculates coordinates of points visible in large and small arrows.
;;; The four points that bound the head of each arrow are the only ones
;;; visible in the small arrows. Points 3 and 4 -- the base of the arrow
;;; -- are obscured, except in the large arrow. We calculate these in
;;; compute-arrow-shaft-points.

```

```

(defun compute-arrowhead-points ()
  (let* ((p1x (- *p0x* *top-edge*)) ;X-coord, point 1
         (p1y *p0y*) ;Y-coord, point 1
         (p2x (+ p1x *top-edge-4*)) ;X-coord, point 2
         (p2y (- *p0y* *top-edge-4*)) ;Y-coord, point 2
         (p6x *p0x*) ;X-coord, point 6
         (p6y (- *p0y* *top-edge*)) ;Y-coord, point 6
         (p5x (- *p0x* *top-edge-4*)) ;X-coord, point 5
         (p5y (+ p6y *top-edge-4*))) ;Y-coord, point 5
    (values p1x p1y p2x p2y p5x p5y p6x p6y)))

```

```

;;; Calculates horizontal distance between stripes.
;;; Distance is a fraction of the distance between stripes for the
;;; large arrow. The divisor depends on the level of recursion.
;;; Distance divides length of top edge evenly when possible to
;;; maintain continuity between head and shaft of arrow.

```

```

(defun compute-stripe-d ()
  ;; Distance should be at least 3 pixels so that there is some
  ;; white space between lines.
  (if (<= *stripe-distance* 3) 3
      ;; First find a fraction of *STRIPE-DISTANCE* that depends
      ;; on recursion level
      (loop for dist = (fixr (/ *stripe-distance*
                               (selectq *depth*
                                     (0 2)
                                     (1 4)
                                     (2 2)
                                     (3 1.5)
                                     (4 1.5)
                                     (otherwise 2))))
            ;; Increment if it doesn't divide *TOP-EDGE* evenly
            then (1+ dist)
            when (= 0 (\ *top-edge* dist))
            ;; Stop when no remainder. Don't return a value
            ;; less than 3.
            do (return (if (<= dist 3) 3 dist))))))

```

```

;;; Calculates the number of lines that compose each stripe.
;;; Calls COMPUTE-DENS to calculate the proportion of distance
;;; between stripes to be filled, then multiplies by the actual
;;; distance between stripes. Makes sure that there is at least
;;; one line and that there aren't too many lines to leave some
;;; white space.
(defun compute-nlines (x)
  ;; Call COMPUTE-DENS and multiply result by *STRIPE-D*
  (let ((n1 (fix (* *stripe-d* (compute-dens x))))
        ;; Supply at least one line
        (cond ((≤ n1 1) 1)
              ;; But leave some white space between lines
              ((≥ n1 (- *stripe-d* 1)) (- *stripe-d* 2))
              (t n1))))

;;; Calculates proportion of distance filled in between each stripe.
;;; The argument is the x-coordinate of the projection of the current
;;; stripe onto the line formed by the top edge. Determines where the
;;; projection of the current stripe is on this line in relation to the
;;; distance from first to last stripes in the arrow. Multiplies this
;;; fraction by the difference between densities of first and last
;;; stripes. Finally, adds the density of the first stripe.
(defun compute-dens (x)
  (+ *d1* (* (- *d2* *d1*)
             (/ (- x *p0x*) (float (- *x2* *p0x*)))))

;;; The following two functions stripe the arrowheads. The
;;; heads of the large and small arrows are identical, so we
;;; use the same functions to stripe both.

```

```

;;; Function controlling striping of the head of each arrow.
;;; Determines coordinates of starting and ending points for each
;;; stripe. Calls COMPUTE-NLINES to determine number of lines for
;;; the stripe. Calls DRAW-ARROWHEAD-LINES to draw the lines that
;;; make up each stripe.

```

```

(defun stripe-arrowhead ()
  ;; Find x-coord of top of last stripe to be drawn
  (loop with last-x = (- *p0x* *top-edge*)
        ;; Find starting x-coord for each stripe, decrementing
        ;; by distance between stripes. Stop at last x-coord.
        for start-x from *p0x* by *stripe-d* above last-x
        ;; Find ending y-coord for each stripe, decrementing by
        ;; distance between stripes.
        for end-y downfrom *p0y* by *stripe-d*
        ;; Find number of lines in the stripe
        for nlines = (compute-nlines start-x)
        ;; Draw the lines that make up the stripe
        do (draw-arrowhead-lines nlines start-x end-y last-x)))

```

```

;;; Draws the lines that make up each stripe in an arrowhead.
;;; Arguments are number of lines in the stripe, starting x-coord
;;; and ending y-coord of first line, and x-coord of top of last
;;; stripe to be drawn. Decrements by one pixel when drawing each
;;; line.

```

```

(defun draw-arrowhead-lines (nlines start-x end-y last-x)
  ;; Set up a counter
  (loop for i from 0 below nlines
        ;; Find starting x-coord, subtracting counter from first
        ;; x-coord
        for first-x = (- start-x i)
        ;; Make sure we don't go past the end of the arrowhead
        while (< last-x first-x)
        ;; Draw a line
        do (send *dest* ':show-lines
                first-x *p0y* *p0x* (- end-y i))))

```

```

;;; The following functions draw and stripe the large arrow

```

```

;;; Function controlling drawing of the large arrow.
;;; Calls functions to find coordinates of vertexes of the arrow.
;;; Outlines the arrow. Binds distance between stripes and x-coord
;;; of projection of last stripe onto top edge. Finally, stripes
;;; head and shaft of arrow when required.
(defun draw-big-arrow ()
  ;; Determine coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Determine coordinates of shaft vertexes
    (multiple-value-bind
      (*p3x* *p3y* *p4x* *p4y*)
      (compute-arrow-shaft-points)
      (draw-big-outline) ;Outline arrow
      (when *do-the-stripes*
        ;; Bind distance between stripes and x-coord of projection
        ;; of last stripe onto top edge
        (let ((*stripe-d* *stripe-distance*)
              (*x2* (- *p0x* *top-edge* *top-edge*)))
          (stripe-arrowhead) ;Stripe head
          (stripe-big-arrow-shaft)))))) ;Stripe shaft

;;; Calculates coordinates for vertexes of shaft of large arrow.
;;; These points are obscured and not drawn for the small arrows.
(defun compute-arrow-shaft-points ()
  (values (- *p1x* *top-edge-4*) ;X-coord of point 3
          (- *p2y* *top-edge-2*) ;Y-coord of point 3
          *p2x* ;X-coord of point 4
          (- *p2y* *top-edge*)) ;Y-coord of point 4

;;; Draws the outline of the large arrow.
(defun draw-big-outline ()
  (send *dest* ':show-lines
        *p0x* *p0y* *p1x* *p1y* *p2x* *p2y* *p3x* *p3y*
        *p4x* *p4y* *p5x* *p5y* *p6x* *p6y* *p0x* *p0y*))

;;; The next seven functions stripe the shaft of the large arrow.
;;; First is a controlling function, then three functions to stripe
;;; the left side and three more to stripe the right.

```



```

;;; Function controlling striping of the shaft of the large arrow.
;;; Just calls STRIPE-BIG-ARROW-SHAFT-LEFT to stripe the left side
;;; and STRIPE-BIG-ARROW-SHAFT-RIGHT to stripe the right side.
(defun stripe-big-arrow-shaft ()
  (stripe-big-arrow-shaft-left)
  (stripe-big-arrow-shaft-right))

;;; Function controlling striping of left side of big arrow's shaft.
;;; Iterates over the triangles that make up the shaft. Determines
;;; coordinates of the apex and bottom right point of each triangle.
;;; Calls DRAW-BIG-ARROW-SHAFT-STRIPES-LEFT to stripe each triangle.
(defun stripe-big-arrow-shaft-left ()
  ;; Set up a counter for depth. Don't exceed maximum recursion
  ;; level.
  (loop for shaft-depth from 0 below *max-depth*
        ;; Find current top edge and its fractions
        for top-edge = *top-edge* then (/ top-edge 2)
        for top-edge-2 = (/ top-edge 2)
        for top-edge-4 = (/ top-edge 4)
        ;; Find coordinates of apex of triangle
        for apex-x = *p2x* then (- apex-x top-edge-2)
        for apex-y = *p2y* then (- apex-y top-edge-2)
        ;; Find x-coord of bottom right vertex
        for right-x = (+ apex-x top-edge-4)
        ;; Find y-coord of bottom edge of triangle
        for bottom-y = (- apex-y top-edge-4)
        ;; Find the x-coord of the projection of the first
        ;; stripe onto top edge
        for xoff = (- *p0x* *top-edge*) then (- xoff top-edge)
        ;; Stripe each triangle
        do (draw-big-arrow-shaft-stripes-left
            top-edge-4 apex-x apex-y right-x bottom-y xoff)))

```

```

;;; Stripes each triangle in left side of big arrow's shaft.
;;; Arguments are one-fourth current top edge, x- and y-coords
;;; of apex of triangle, x- and y-coords of bottom right vertex,
;;; and x-coord of projection of first stripe onto top edge.
;;; Determines coordinates of starting and ending points for
;;; each stripe. Finds number of lines in the stripe. Calls
;;; DRAW-BIG-ARROW-SHAFT-LINES-LEFT to draw the lines that
;;; make up each stripe.
(defun draw-big-arrow-shaft-stripes-left
  (top-edge-4 apex-x apex-y right-x bottom-y xoff)
  (loop with half-distance = (// *stripe-distance* 2)
    ;; Find x-coord of last stripe in triangle
    with last-x = (- apex-x top-edge-4)
    ;; Find x-coord of top of each stripe, decrementing
    ;; from the apex by HALF the horizontal distance
    ;; between stripes. Stop at last stripe.
    for start-x from apex-x by half-distance above last-x
    ;; Find y-coord of top of stripe
    for start-y downfrom apex-y by half-distance
    ;; Find x-coord of endpoint of stripe
    for end-x downfrom right-x by *stripe-distance*
    ;; Find number of lines in the stripe
    for nlines = (compute-nlines (- xoff (- right-x end-x)))
    ;; Draw a stripe
    do (draw-big-arrow-shaft-lines-left
        nlines start-x start-y end-x bottom-y last-x)))

```

```
;;; Draws the lines for a stripe on left side of big arrow's shaft.
;;; Arguments are number of lines in the stripe, coords of starting
;;; and ending points for first line, and x-coord of last stripe to
;;; be drawn.
(defun draw-big-arrow-shaft-lines-left
  (nlines start-x start-y end-x end-y last-x)
  ;; Set up two counters -- we need to draw two lines at once
  (loop for i from 0
        for i2 from 0 by 2
        ;; Find x-coord of top of first line in stripe
        for first-x = (- start-x i)
        ;; Don't exceed number of lines in stripe
        while (< i2 nlines)
        ;; Don't go past the end of the triangle
        while (< last-x first-x)
        ;; Draw a line
        do (send *dest* ':show-lines first-x (- start-y i)
                (- end-x i2) end-y)
        ;; Draw a second line. The two lines are a refinement
        ;; to stagger the endpoints of the lines so the diagonal
        ;; edge looks neat.
        (send *dest* ':show-lines first-x (- start-y i 1)
                (- end-x i2 1) end-y)))
```

```
;;; Function controlling striping of right side of big arrow's shaft.
;;; Iterates over the triangles that make up the shaft. Determines
;;; coordinates of the top point of each triangle. Calls
;;; DRAW-BIG-ARROW-SHAFT-STRIPES-RIGHT to stripe each triangle.
(defun stripe-big-arrow-shaft-right ()
  ;; Set up a counter for depth. Don't exceed maximum recursion
  ;; level.
  (loop for shaft-depth from 0 below *max-depth*
        ;; Find new top edge and its fractions
        for top-edge = *top-edge* then (/ top-edge 2)
        for top-edge-2 = (/ top-edge 2)
        for top-edge-4 = (/ top-edge 4)
        ;; Find coords of top point of triangle
        for start-x = (+ *p2x* top-edge-4)
        for top-y = (- *p2y* *top-edge-4*)
        then (- top-y top-edge-2 top-edge-4)
        ;; Find x-coord of projection of first stripe onto
        ;; top-edge
        for xoff = (- *p0x* *top-edge*) then (- xoff top-edge)
        ;; Stripe the triangle
        do (draw-big-arrow-shaft-stripes-right
            top-edge-2 top-edge-4 start-x top-y xoff)))
```

```
;;; Stripes each triangle in right side of big arrow's shaft.
;;; Arguments are one-half and one-fourth of current top edge,
;;; coords of top point of the triangle, and x-coord of projection
;;; of first stripe onto top edge. Determines coordinates of
;;; starting and ending points for each stripe. Finds number of
;;; lines that make up the stripe. Calls
;;; DRAW-BIG-ARROW-SHAFT-LINES-RIGHT to draw a stripe.
(defun draw-big-arrow-shaft-stripes-right
  (top-edge-2 top-edge-4 start-x top-y xoff)
  (loop with half-distance = (/ *stripe-distance* 2)
        ;; Find y-coord of last stripe in triangle
        with last-y = (- top-y top-edge-2)
        ;; Find y-coord of starting point of stripe. Don't go
        ;; past the end of the triangle.
        for start-y from top-y by *stripe-distance* above last-y
        ;; Find coords of ending point of the stripe, decrementing
        ;; by HALF the horizontal distance between stripes
        for end-x downfrom (+ start-x top-edge-4) by half-distance
        for end-y downfrom (- top-y top-edge-4) by half-distance
        ;; Find number of lines that make up the stripe
        for nlines = (compute-nlines (- xoff (- top-y start-y)))
        ;; Draw a stripe
        do (draw-big-arrow-shaft-lines-right
            nlines start-x start-y end-x end-y last-y)))
```

```
;;; Draws the lines for a stripe on right side of big arrow's shaft.
;;; Arguments are number of lines in the stripe, coordinates of starting
;;; and ending points for the first line, and y-coord of last stripe in
;;; the triangle.
(defun draw-big-arrow-shaft-lines-right
  (nlines start-x start-y end-x end-y last-y)
  ;; Set up two counters -- we need to draw two lines at once
  (loop for i from 0
        for i2 from 0 by 2
        ;; Find y-coord of ending point of line
        for stop-y = (- end-y i)
        ;; Don't exceed number of lines in the stripe
        while (< i2 nlines)
        ;; Don't go past the bottom of the triangle
        while (< last-y stop-y)
        ;; Draw a line
        do (send *dest* ':show-lines start-x (- start-y i2)
                (- end-x i) stop-y)
        ;; Draw a second line. The two lines are a refinement
        ;; to stagger the endpoints of the lines so the diagonal
        ;; edge looks neat.
        (send *dest* ':show-lines start-x (- start-y i2 1)
                (- end-x i 1) stop-y)))

;;; The remaining functions draw and stripe one of the small arrows
```

```

;;; Function controlling drawing of a small arrow.
;;; Calculates coordinates of the arrowhead and outlines it. Binds x-coord
;;; of the projection of the last stripe onto the top edge. Calculates
;;; the horizontal distance between stripes. When necessary, stripes the
;;; head and shaft of the arrow.
(defun draw-arrow ()
  ;; Calculate coordinates of arrowhead vertexes
  (multiple-value-bind
    (*p1x* *p1y* *p2x* *p2y* *p5x* *p5y* *p6x* *p6y*)
    (compute-arrowhead-points)
    ;; Outline the arrowhead
    (draw-outline)
    (when *do-the-stripes*
      ;; Bind x-coord of projection of last stripe onto top edge
      (let ((*x2* (- *p0x* *top-edge* *top-edge*)))
        ;; Calculate distance between stripes
        (*stripe-d* (compute-stripe-d)))
        (stripe-arrowhead)                ;Stripe head
        (stripe-arrow-shaft))))))        ;Stripe shaft

;;; Draws the outline of the head of a small arrow.
(defun draw-outline ()
  (send *dest* ':show-lines *p2x* *p2y* *p1x* *p1y*
    *p0x* *p0y* *p6x* *p6y* *p5x* *p5y*))

```

```
;;; Function controlling striping of the shaft of a small arrow.
;;; Iterates over the descending triangles that make up the shaft.
;;; Calculates the coordinates of the top left and bottom right
;;; vertexes of each triangle. Finds the x-coord of the
;;; projection of the first stripe onto top edge. Calls
;;; DRAW-ARROW-SHAFT-STRIPES to stripe each triangle.
(defun stripe-arrow-shaft ()
  ;; Set up a counter for depth. Don't exceed maximum
  ;; recursion level.
  (loop for shaft-depth from *depth* below *max-depth*
        ;; Calculate fractions of new top edge
        for top-edge-2 = *top-edge-2* then (/ top-edge-2 2)
        for top-edge-4 = (/ top-edge-2 2)
        ;; Find coords of top left point of triangle
        for left-x = *p2x* then (- left-x top-edge-4)
        for top-y = *p2y* then (- top-y top-edge-2 top-edge-4)
        ;; Find coords of bottom right point of triangle
        for right-x = (+ left-x top-edge-2)
        for bottom-y = (- top-y top-edge-2)
        ;; Find x-coord of projection of first stripe onto top edge
        for xoff = (- *p0x* *top-edge*)
        then (- xoff top-edge-2 top-edge-2)
        ;; Stripe the triangle
        do (draw-arrow-shaft-stripes
           left-x top-y right-x bottom-y xoff)))
```



```

;;; Stripes each triangle in the shaft of a small arrow.
;;; Arguments are coordinates of the top left and bottom right
;;; points of the triangle, and the x-coord of the projection
;;; of the first stripe onto top edge. Calculates the y-coord
;;; of the starting point and the x-coord of the ending point
;;; of each stripe. Finds number of lines in the stripe. Calls
;;; DRAW-ARROW-SHAFT-LINES to draw the lines in the stripe.
(defun draw-arrow-shaft-stripes
  (left-x top-y right-x bottom-y xoff)
  ;; Find y-coord of starting point of stripe. Don't go
  ;; below the bottom of the triangle.
  (loop for start-y from top-y by *stripe-d* above bottom-y
        ;; Find x-coord of ending point of the stripe
        for end-x downfrom right-x by *stripe-d*
        ;; Find number of lines in the stripe
        for nlines = (compute-nlines (- xoff (- right-x end-x)))
        ;; Draw a stripe
        do (draw-arrow-shaft-lines
            nlines left-x start-y end-x bottom-y)))

;;; Draws the lines in a stripe in the shaft of a small arrow.
;;; Arguments are the number of lines in the stripe and the
;;; coordinates of the starting and ending points of the first line.
(defun draw-arrow-shaft-lines
  (nlines left-x start-y end-x bottom-y)
  ;; Set up a counter. Don't exceed number of lines in the stripe.
  (loop for i from 0 below nlines
        ;; Find x-coord of ending point of the line
        for last-x = (- end-x i)
        ;; Don't go past the left edge of the triangle
        while (< left-x last-x)
        ;; Draw a line
        do (send *dest* ':show-lines left-x (- start-y i)
                last-x bottom-y)))

```

### 33. Output Module for the Sample Program

The program used as an example in this document draws the recursive arrow graphic on the document's cover. This section contains Lisp code that defines the flavors and methods that mediate between the program and the system output operations. For the code that calculates coordinates for the endpoints of the lines that compose the figure: See the section "Calculation Module for the Sample Program", page 383. For a reproduction of the LGP graphic the program produces: See the section "Graphic Output of the Sample Program", page 425.

```
;;; -*- Mode: LISP; Package: (GRAPHICS GLOBAL 1000); Base: 10 -*-  
;;; Copyright (c) 1983 Symbolics, Inc.
```

```
#||
```

This file contains the output module for a program that reproduces the recursive arrow graphic printed on the covers of most Symbolics documents. The module allows the graphic to be produced on a Lisp Machine screen, a Laser Graphics Printer, or an LGP record file. For each of these devices, the module produces output by sending appropriate messages with the coordinates of the endpoints of line segments to be drawn. This module receives these coordinates from a separate calculation module.

For screen output, the module creates its own windows. It defines a basic flavor of window that accepts point coordinates in the screen coordinate system, with origin at top left. It defines a more specialized window, built on the basic window, for use with a calculation module that uses LGP coordinates, with origin at bottom left. It allows a process to be associated with each window and lets users modify the characteristics of the figure.

For LGP output, the module makes an instance of a flavor with the output stream as an instance variable. Output is directed to either a hardcopy device or a record file.

This module defines the top-level function, DO-ARROW, that is called to produce the graphic. This function pops up a choose-variable-values window to allow users to select the output device and the characteristics of the figure. The module defines conditions and handlers for attempts to give variables impermissible values.

This module determines the size of the figure and its position within the page or window. It then calls the function DRAW-ARROW-GRAPHIC in the calculation module. It passes as arguments the length of the top edge of the figure and the coordinates of the top right point. The calculation module sends :SHOW-LINES messages to instances of output flavors. The arguments to :SHOW-LINES are the coordinates of the endpoints of lines to be drawn. The current instance of the output flavor is the value of the special variable \*DEST\*.

```
||#
```

```
;;; Following are declarations for special variables
```

```
(defvar *dest-string* "Screen"  
  "Destination of program output [Screen, LGP, or File]")
```

```
(defvar *output-file* nil  
  "Pathname for LGP-record-file output")
```

```
(defvar *fill-proportion* 0.9  
  "Proportion of smaller dimension to be filled by figure")
```

```
;;; The following flavor and its methods are common to both  
;;; screen and LGP output
```

```

(defflavor arrow-parameter-mixin
  (width height top-edge right-x top-y)
  ())
  (:gettable-instance-variables top-edge right-x top-y)
  (:required-methods :compute-width-and-height)
  (:documentation :mixin
    "Provides parameters for size and position of figure.
Instance variables hold width and height of page or window;
length of top edge of figure; and coordinates of top right point
of figure. Methods calculate size and position of figure by
centering it within the page or window and making it fill no
more than the specified proportion of the smaller dimension.
The methods use a coordinate system with origin at bottom left;
other mixins must correct for this if output is going to a
window. Other flavors must also provide a method for calculating
width and height of the page or window. This flavor should be
mixed into any instantiable flavor that produces output for the
arrow graphic."))

;;; Method controlling calculation of size and position of figure.
;;; Sends messages to self to calculate width and height of page
;;; or window, length of top edge of figure, and coordinates of
;;; figure's top right point. These are separate methods so that
;;; other flavors can shadow them or add daemons. Another flavor
;;; must provide a method to compute width and height, because
;;; this is specific to the output device.
(defmethod (arrow-parameter-mixin :compute-parameters) ()
  ;; Another flavor must supply method for width and height
  (send self ':compute-width-and-height)
  ;; Make a preliminary estimate of length of top edge
  (send self ':compute-top-edge)
  ;; Adjust top edge to make it a multiple of 128
  (send self ':adjust-top-edge)
  ;; Calculate coordinates of top right point of figure.
  ;; We can't do this until we know how long top edge is.
  (send self ':compute-right-x)
  (send self ':compute-top-y))

```

```
;;; Makes a preliminary estimate of length of top edge.
;;; The top edge of the arrow is 80 percent of the horizontal
;;; or vertical length of the whole figure. First finds the
;;; smaller of the length or width of the page or window.
;;; Multiplies this by the proportion of this dimension that
;;; is to be filled by the figure. The result is the
;;; horizontal or vertical length of the figure. Multiplies
;;; this by 0.8 to get the length of the top edge.
(defmethod (arrow-parameter-mixin :compute-top-edge) ()
  (setq top-edge
    (fixr (* 0.8 *fill-proportion* (min width height))))))
```

```
;;; Adjusts length of top edge so it is a multiple of 128.
;;; There are 64 stripes in the head of the large arrow. The
;;; calculation module divides the length of top edge by two
;;; each time it goes down another recursion level. By making
;;; the original top edge a multiple of 128, we maximize
;;; continuity in striping between arrowheads and shafts and
;;; among the first several levels of recursion.
(defmethod (arrow-parameter-mixin :adjust-top-edge) ()
  (setq top-edge
    ;; Minimum length of top edge is 128
    (if (< top-edge 256) 128
        ;; Otherwise set to next lower multiple of 128
        (* 128 (fix (// top-edge 128)))))
```

```
;;; Calculates x-coordinate of top right point of figure.
;;; Finds horizontal length of figure by dividing length of
;;; top edge by 0.8. Centers the figure horizontally within
;;; the page or window.
(defmethod (arrow-parameter-mixin :compute-right-x) ()
  (setq right-x
    (fixr (* 0.5 (+ width (// top-edge 0.8)))))
```

```
;;; Calculates y-coordinate of top right point of figure.
;;; Assumes that the origin is at bottom. Finds vertical
;;; length of figure by dividing length of top edge by 0.8.
;;; Centers the figure vertically within the page or window.
(defmethod (arrow-parameter-mixin :compute-top-y) ()
  (setq top-y
    (fixr (* 0.5 (+ height (// top-edge 0.8)))))
```

;;; Following are flavors and methods for screen output

```
(defflavor basic-arrow-window-mixin
  (do-stripes max-dep fill-prop)
  ()
  :initable-instance-variables
  (:required-flavors arrow-parameter-mixin tv:window)
  (:default-init-plist
   :edges-from 'mouse :minimum-width 200 :minimum-height 200
   :blinker-p nil :expose-p t)
  (:documentation :mixin
   "Provides for a basic window to display the arrow graphic.
  ARROW-PARAMETER-MIXIN is needed to position the figure within
  the window. Instance variables hold values for maximum
  recursion level, proportion of window to be filled, and
  whether or not to stripe the figure. This flavor assumes
  window coordinates, with origin at top left. It provides its
  own :COMPUTE-TOP-Y method to use that origin. It provides a
  method to find the width and height of the window, as
  ARROW-PARAMETER-MIXIN requires. This flavor has a :SHOW-LINES
  method to receive point coordinates from the calculation
  module and draw lines on the window. It provides a :MAIN-LOOP
  method so that the window can run in its own process and let
  the user modify the graphic. TV:LIST-MOUSE-BUTTONS-MIXIN is
  needed to handle mouse clicks if this method is used. This
  flavor provides standard :AFTER daemons for the window-system
  :INIT, :REFRESH, and :CHANGE-OF-SIZE-OR-MARGINS messages. This
  flavor should be mixed in with ARROW-PARAMETER-MIXIN and
  TV:WINDOW for any window that produces the graphic. It
  should be included before ARROW-PARAMETER-MIXIN so that the
  :COMPUTE-TOP-Y method shadows correctly."))
```

```

;;; Receives endpoint coordinates and draws lines on a window.
;;; Arguments are alternating x- and y-coordinates of the end-
;;; points of lines to be drawn. If there are more than two pairs
;;; of coordinates, assumes that the endpoint of one line is the
;;; starting point of the next. Sends messages for separate methods
;;; to determine the actual coordinates. This is so that other
;;; flavors can modify the coordinates. Draws a line by sending self
;;; a :DRAW-LINE message, and so assumes that TV:GRAPHICS-MIXIN is
;;; included somewhere to provide this method.

```

```

(defmethod (basic-arrow-window-mixin :show-lines)
  (x y &rest x-y-pairs)
  ;; First determine the starting point of the line. On
  ;; subsequent trips through the loop, the last endpoint
  ;; becomes the next starting point.
  (loop for x0 = (send self ':compute-x x) then x1
        for y0 = (send self ':compute-y y) then y1
        ;; "Cddr" down the list created by making all but the
        ;; first pair of coordinates an &rest argument
        for (x1 y1) on x-y-pairs by #'cddr
        ;; Determine the endpoint of the line
        do (setq x1 (send self ':compute-x x1)
              y1 (send self ':compute-y y1))
        ;; Draw the line
        (send self ':draw-line
                 x0 y0 x1 y1 tv:alu-ior t)))

```

```

;;; Determines the x-coordinate of an endpoint of a line.
;;; This is a separate method so that other flavors can shadow
;;; it or add daemons to manipulate the coordinate.

```

```

(defmethod (basic-arrow-window-mixin :compute-x) (x)
  (fixr x))

```

```

;;; Determines the y-coordinate of an endpoint of a line.
;;; Assumes that the argument already uses window coordinates,
;;; with origin at top left. This is a separate method so that
;;; other flavors can shadow it or add daemons to manipulate
;;; the coordinate.

```

```

(defmethod (basic-arrow-window-mixin :compute-y) (y)
  (fixr y))

```

```

;;; Finds the inside width and height of the window.
;;; Sends self an :INSIDE-SIZE message, and so assumes that
;;; TV:SHEET is included somewhere to provide this
;;; method.
(defmethod (basic-arrow-window-mixin
           :compute-width-and-height) ()
  (multiple-value (width height)
    (send self ':inside-size)))

;;; Calculates y-coordinate of top right point of figure.
;;; Finds vertical length of the figure by dividing the length
;;; of top edge by 0.8. Centers the figure vertically within
;;; the window. Gives the result in window coordinates, with
;;; origin at top left. This method shadows that in
;;; ARROW-PARAMETER-MIXIN.
(defmethod (basic-arrow-window-mixin :compute-top-y) ()
  (setq top-y
    (fixr (* 0.5 (- height (/ top-edge 0.8))))))

;;; Calculates size and position of figure after initialization.
;;; Binds the global variable *fill-proportion* to the value of
;;; the corresponding instance variable so that the figure will
;;; be drawn correctly if the value of *fill-proportion* has
;;; changed.
(defmethod (basic-arrow-window-mixin :after :init) (ignore)
  (let ((*fill-proportion* fill-prop))
    (send self ':compute-parameters)))

;;; Calculates size and position of figure after window change.
;;; Binds the global variable *fill-proportion* to the value of
;;; the corresponding instance variable so that the figure will
;;; be drawn correctly if the value of *fill-proportion* has
;;; changed.
(defmethod (basic-arrow-window-mixin
           :after :change-of-size-or-margins) (&rest ignore)
  (let ((*fill-proportion* fill-prop))
    (send self ':compute-parameters)))

```



```

;;; Draws the figure when necessary after window is refreshed.
;;; Binds the global variable *dest* to self and the variables
;;; *do-the-stripes* and *max-depth* to the corresponding instance
;;; variables so the figure will be drawn correctly if the values
;;; of the global variables have changed.
(defmethod (basic-arrow-window-mixin :after :refresh)
  (&optional type)
  ;; Draw figure if not restored from a bit-save array ...
  (when (or (not tv:restored-bits-p)
            ;; ... or size has changed ...
            (eq type ':size-changed)
            ;; ... or new values for figure parameters.
            (eq type ':new-vals))
    ;; If restored from a bit-save array, clear screen first
    (when tv:restored-bits-p
      (send self ':clear-screen))
    ;; Bind global variables to self and instance variables
    (let ((*dest* self)
          (*do-the-stripes* do-stripes)
          (*max-depth* max-dep))
      ;; Draw the figure
      (draw-arrow-graphic top-edge right-x top-y))))

;;; Provides a mouse documentation line for the window.
;;; The only option is to click right and pop up a
;;; choose-variable-values window of options for changing
;;; the graphic on this window.
(defmethod (basic-arrow-window-mixin
           :who-line-documentation-string) ()
  "R: Choose-variable-values options for changing figure on this window")

```

```

;;; Command loop for window associated with a separate process.
;;; Consists of an error-restart-loop that handles restarts from
;;; errors and sys:abort. Waits for mouse input. If a right
;;; click, pops up a choose-variable-values window to change
;;; characteristics of the figure. On exit, sets instance variables
;;; to the new values and refreshes the window, then waits for another
;;; mouse click. Assumes blips are lists of the form provided
;;; by TV:LIST-MOUSE-BUTTONS-MIXIN.
(defmethod (basic-arrow-window-mixin :main-loop) ()
  ;; Run forever in a loop. Offer a restart handler if an error
  ;; or sys:abort is signalled.
  (error-restart-loop ((error sys:abort) "Arrow Window Top Level")
    ;; Wait for input
    (let ((char (send self ':any-tyi)))
      ;; Pop up window if input is a list ...
      (when (and (listp char)
                 ;; ... and a mouse click ...
                 (eq (first char) ':mouse-button)
                 ;; ... and a single click on the right button.
                 (eq (second char) #\mouse-r-1))
        ;; Bind global variables to instance-variable values
        (let ((*do-the-stripes* do-stripes)
              (*max-depth* max-dep)
              (*fill-proportion* fill-prop))
          ;; Pop up a choose-variable-values window
          (tv:choose-variable-values
            '(((*do-the-stripes* "Stripe the arrows?" :boolean)
              (*max-depth* "Number of recursion levels" :number)
              (*fill-proportion*
                "Fraction of window to be filled" :number))
            ;; Make the window wide to provide enough room for error
            ;; messages.
            ':extra-width 20
            ;; Call a function to check for errors when values change
            ':function 'check-item
            ;; Give the user a chance to abort
            ':margin-choices '("Do It" ("Abort" (signal 'sys:abort)))
            ':label "Choose Options for Graphic")
          ;; Set instance variables to the new values
          (setq do-stripes *do-the-stripes*
                max-dep *max-depth*
                fill-prop *fill-proportion*))

```

```
;; Recompute size and position of the figure
(send self ':compute-parameters)
;; Send :REFRESH message with argument of ':new-vals to make
;; sure the figure is redrawn if there is a bit-save array
(send self ':refresh ':new-vals))))))
```

```
(defflavor basic-arrow-window ()
  (basic-arrow-window-mixin
   arrow-parameter-mixin
   tv:any-tyi-mixin
   tv:list-mouse-buttons-mixin
   tv:process-mixin
   tv>window)
  (:documentation :combination
  "Instantiable flavor providing a basic window for output.
  Though this flavor is instantiable, its methods assume that
  point coordinates use the window coordinate system, with
  origin at top left. To work with the current calculation
  module it needs another mixin to convert LGP to screen
  coordinates. In the component flavors, BASIC-ARROW-WINDOW-MIXIN
  must come before ARROW-PARAMETER-MIXIN and TV:WINDOW for
  shadowing and daemons to work correctly. TV:PROCESS-MIXIN
  and TV:LIST-MOUSE-BUTTONS-MIXIN are not necessary unless the
  window is associated with a separate process and the :MAIN-LOOP
  method of BASIC-ARROW-WINDOW-MIXIN is the command loop."))
```

```

(defflavor lgp-window-mixin
  ((scale-factor 2.5))
  ()
  (:required-flavors basic-arrow-window)
  (:documentation :mixin
    "Converts LGP to screen coordinates and vice versa.
    When mixed in with BASIC-ARROW-WINDOW, this flavor allows
    window output with a calculation module that uses LGP
    coordinates. The instance variable SCALE-FACTOR is the
    ratio of LGP to screen pixel density. The methods take
    the height and width of the window in screen pixels and
    calculate the length of the top edge and the coordinates
    of the top right point of the figure in LGP pixels. In
    drawing lines on the window, the methods convert LGP to
    window coordinates. These methods shadow those in
    ARROW-PARAMETER-MIXIN and BASIC-ARROW-WINDOW-MIXIN.")
  ))

;;; Converts x-coord of line endpoint from LGP to screen pixels.
;;; Corrects for higher density of LGP pixels. This method shadows
;;; that of BASIC-ARROW-WINDOW-MIXIN.
(defmethod (lgp-window-mixin :compute-x) (x)
  (fixr (/ x scale-factor)))

;;; Converts y-coord of line endpoint from LGP to screen pixels.
;;; Corrects for higher density of LGP pixels and for screen origin
;;; at top left. This method shadows that of BASIC-ARROW-WINDOW-MIXIN.
(defmethod (lgp-window-mixin :compute-y) (y)
  (fixr (- height (/ y scale-factor))))

;;; Calculates top edge in LGP pixels from screen proportions.
;;; Multiplies length of smaller dimension, in screen pixels, by
;;; proportion of this dimension to be filled by the figure.
;;; Multiplies this by 0.8 to find top edge in screen pixels.
;;; Corrects for higher density of LGP pixels. This method
;;; shadows that of ARROW-PARAMETER-MIXIN.
(defmethod (lgp-window-mixin :compute-top-edge) ()
  (setq top-edge
    (fixr (* scale-factor 0.8 *fill-proportion*
      (min width height)))))

```

```

;;; Calculates x-coord of top right point in LGP pixels.
;;; Finds horizontal length of figure in screen pixels by
;;; dividing top edge by 0.8. Centers figure horizontally
;;; in window, correcting for higher density of LGP pixels.
;;; This method shadows that of ARROW-PARAMETER-MIXIN.
(defmethod (lgp-window-mixin :compute-right-x) ()
  (setq right-x
    (fixr (* 0.5 (+ (* width scale-factor)
                    (/ top-edge 0.8))))))

;;; Calculates y-coord of top right point in LGP pixels.
;;; Finds vertical length of figure in screen pixels by
;;; dividing top edge by 0.8. Centers figure vertically
;;; in window, correcting for higher density of LGP pixels.
;;; This method shadows those of ARROW-PARAMETER-MIXIN and
;;; BASIC-ARROW-WINDOW-MIXIN.
(defmethod (lgp-window-mixin :compute-top-y) ()
  (setq top-y
    (fixr (* 0.5 (+ (* height scale-factor)
                    (/ top-edge 0.8))))))

(defflavor arrow-window ()
  (lgp-window-mixin basic-arrow-window)
  (:documentation :combination
   "Instantiable flavor for window output from LGP coordinates.
This flavor has all the features of BASIC-ARROW-WINDOW but
assumes that the calculation module uses LGP coordinates. This
is the flavor to instantiate for window output using the
current calculation module."))

;;; The following flavor and methods are for LGP output

```

```
(defflavor lgp-pixel-mixin
  (output-stream)
  ()
  :initable-instance-variables
  (:required-flavors arrow-parameter-mixin)
  (:documentation :mixin
    "Provides methods for arrow graphic output on an LGP stream.
    ARROW-PARAMETER-MIXIN is required to calculate the size of the
    figure and position it in the center of the page. This flavor
    has a method to calculate the width and height of the page, as
    ARROW-PARAMETER-MIXIN requires. It has a :SHOW-LINES method to
    receive point coordinates from the calculation module and draw
    lines on the output stream. The method assumes that coordinates
    are in LGP pixels. The method also assumes that flavor
    LGP:BASIC-LGP-STREAM is included in output stream to provide
    :SEND-COMMAND and :SEND-COORDINATES messages. This flavor
    should be mixed, along with ARROW-PARAMETER-MIXIN, into an
    instantiable flavor for LGP output. When that flavor is
    instantiated, the instance variable output-stream should be
    initialized."))
```

```
;;; Receives endpoint coordinates and draws lines on LGP stream.
;;; Arguments are alternating x- and y-coordinates of endpoints of
;;; lines to be drawn. If there are more than two pairs of
;;; coordinates, assumes that the endpoint of one line is the
;;; starting point of the next. Draws a line by sending output
;;; stream :SEND-COMMAND messages for LGP commands and
;;; :SEND-COORDINATE messages for LGP coordinates. Assumes that
;;; flavor LGP:BASIC-LGP-STREAM is included in output stream to
;;; provide these methods.
```

```
(defmethod (lgp-pixel-mixin :show-lines)
  (x0 y0 &rest x-y-pairs)
  ;; Send command and coordinates to start drawing lines
  (send self ':send-command-and-coordinates #/m x0 y0)
  ;; "Cddr" down the list created by making all but the first
  ;; pair of coordinates an &rest argument
  (loop for (x y) on x-y-pairs by #'cddr
    ;; Send command and coordinates to draw a line
    do (send self ':send-command-and-coordinates #/v x y)))
```

```

;;; Sends line-drawing commands to LGP output stream.
;;; :SEND-COMMAND transmits an LGP command. :SEND-COORDINATES
;;; transmits coordinates of an endpoint of a line to be drawn.
;;; Assumes that LGP:BASIC-LGP-STREAM is included in output stream
;;; to provide these methods.
(defmethod (lgp-pixel-mixin :send-command-and-coordinates) (cmd x y)
  (send output-stream ':send-command cmd)
  (send output-stream ':send-coordinates (fixr x) (fixr y)))

;;; Finds width and height of a page for LGP output.
;;; This flavor is required by ARROW-PARAMETER-MIXIN. Finds the
;;; values of two instance variables of LGP:BASIC-LGP-STREAM:
;;; SI:PAGE-WIDTH and SI:PAGE-HEIGHT. Assumes that
;;; LGP:BASIC-LGP-STREAM is included in output stream to provide
;;; these instance variables.
(defmethod (lgp-pixel-mixin :compute-width-and-height) ()
  (setq width (symeval-in-instance output-stream 'si:page-width)
        height (symeval-in-instance output-stream 'si:page-height)))

(defflavor lgp-pixel-stream ()
  (lgp-pixel-mixin arrow-parameter-mixin)
  (:documentation :combination
   "Instantiable flavor for arrow output on LGP stream.
Assumes that the calculation module uses LGP coordinates.
When this flavor is instantiated, the LGP-PIXEL-MIXIN
instance variable OUTPUT-STREAM should be initialized.
The output stream can be directed to an LGP or a file,
but it must include flavor LGP:BASIC-LGP-STREAM for
output to work correctly."))

;;; Following are condition flavors for bad variable values

```

```

(defflavor bad-arrow-variable () (error)
  (:documentation
    "Noninstantiable class of bad-variable conditions.
    The user might set some variables to impermissible values.
    These conditions are to permit checking for bad values
    beyond the system's error checking. Instantiable condition
    flavors for specific variables should be built on this
    flavor."))

(defflavor bad-arrow-depth () (bad-arrow-variable)
  (:documentation
    "Proceedable condition: bad value for *MAX-DEPTH*.
    An instantiable condition flavor for impermissible values
    of *MAX-DEPTH*, the number of recursion levels in the
    figure."))

;;; Prints string on stream to report bad *MAX-DEPTH* value
(defmethod (bad-arrow-depth :report) (stream)
  (format stream "No. of levels was not a ~
                nonnegative fixnum."))

;;; Proceed type method for supplying new value of *MAX-DEPTH*
(defmethod (bad-arrow-depth :case :proceed :new-depth)
  (&optional (dep (prompt-and-read
                    ':number
                    "Supply new value for ~
                    no. of recursion levels: ")))
  "Supply a new value for number of recursion levels."
  (values ':new-depth dep))

(defflavor bad-arrow-fill-proportion () (bad-arrow-variable)
  (:documentation
    "Proceedable condition: bad value for *FILL-PROPORTION*.
    An instantiable condition flavor for impermissible values of
    *FILL-PROPORTION*, the fraction of the smaller dimension of
    the page or window that the figure is to fill."))

```



```
;;; Prints string on stream to report bad *FILL-PROPORTION* value.
(defmethod (bad-arrow-fill-proportion :report) (stream)
  (format stream "Proportion was not a fraction between ~
                0 and 1.~"))

;;; Proceed type method for new value of *FILL-PROPORTION*
(defmethod (bad-arrow-fill-proportion :case :proceed
                                       :new-proportion)
  (&optional (prop (prompt-and-read
                    ':number
                    "Supply new fraction of bounds ~
                    be filled: ~"))))
  "Supply a new fraction of page or window to be filled."
  (values ':new-proportion prop))

;;; Top-level function
```

```
;;; Top-level function to call to produce arrow graphic.
;;; Pops up a choose-variable-values window to let user specify
;;; output destination, number of recursion levels, proportion
;;; of smaller dimension of page or window to be filled, and
;;; whether or not to stripe figure. If screen output, makes a
;;; window. If LGP output, makes an LGP stream and calls
;;; DRAW-ARROW-GRAPHIC to draw the figure.
(defun do-arrow ()
  ;; Pop up a choose-variable-values window
  (tv:choose-variable-values
   '(((*do-the-stripes* "Stripe the arrows?" :boolean)
     (*max-depth* "Number of recursion levels" :number)
     (*fill-proportion*
      "Fraction of page or window to be filled" :number)
     (*dest-string* "Output destination"
      :choose ("Screen" "LGP" "File"))
     (*output-file* "Pathname for file output" :pathname))
   ;; Make window wide enough to accommodate long pathnames
   ;; and error messages
   ':extra-width 20.
   ;; Call this function when a value is changed
   ':function 'check-item
   ;; Give user a chance to abort
   ':margin-choices '("Do It" ("Abort" (signal 'sys:abort)))
   ':label "Choose Options for Graphic")
```



```

;;; Top-level function for process associated with arrow window.
;;; The function is called when the window is created. Argument is
;;; the window. The function sends the window a :MAIN-LOOP message.
;;; This method should be the actual command loop for the process.
(defun window-loop (window)
  (send window ':main-loop))

;;; Function to check variable values

;;; Called when a value changes in choose-variable-values window.
;;; Arguments are the window, the variable, and its old and new values.
;;; Binds handlers for conditions for impermissible values. If new
;;; value is OK, sets variable to the new value, refreshes window, and
;;; returns t. If value is not OK, signals the appropriate condition.
;;; When SIGNAL returns, presumably with a new variable value, checks
;;; the new value in the same way it checks a new value that comes
;;; from the window.
(defun check-item (cvv-window var old-val new-val)
  ;; We don't use the old value. To avoid a compiler complaint,
  ;; just evaluate it and ignore it. We could also use IGNORE
  ;; instead of OLD-VAL in the arglist, but then the arglist
  ;; would be less meaningful.
  old-val
  ;; Bind handlers for the conditions we might signal
  (condition-bind ((bad-arrow-depth 'bad-arrow-var-handler)
                  (bad-arrow-fill-proportion
                   'bad-arrow-var-handler))
    (when (eq var '*max-depth*)
      ;; *MAX-DEPTH* must be nonnegative fixnum
      (loop until (and (fixp new-val) (>= new-val 0))
        ;; If it's not, bind QUERY-IO to the window and
        ;; signal a condition. SIGNAL should return
        ;; two values, the proceed type and the new
        ;; value from the proceed method. Ignore the
        ;; proceed type and set NEW-VAL to the new
        ;; value.
        do (let ((query-io cvv-window))
            (multiple-value (nil new-val)
              (signal 'bad-arrow-depth))))))

```

```
(when (eq var '*fill-proportion*)
  ;; *FILL-PROPORTION* must be between 0 and 1
  (loop until (and ( $\geq$  new-val 0) ( $\leq$  new-val 1))
    ;; If it's not, bind QUERY-IO to the window and
    ;; signal a condition. SIGNAL should return
    ;; two values, the proceed type and the new
    ;; value from the proceed method. Ignore the
    ;; proceed type and set NEW-VAL to the new
    ;; value.
    do (let ((query-io cvv-window)
            (multiple-value (nil new-val)
              (signal 'bad-arrow-fill-proportion))))
      ;; Variable value is now OK. Set variable to the new value.
      ;; Note that we DO want to evaluate VAR.
      (set var new-val)
      ;; Refresh the window
      (send cvv-window ':refresh)
      ;; Return t
      t))

;;; Handler for bad-variable-value conditions
```

```

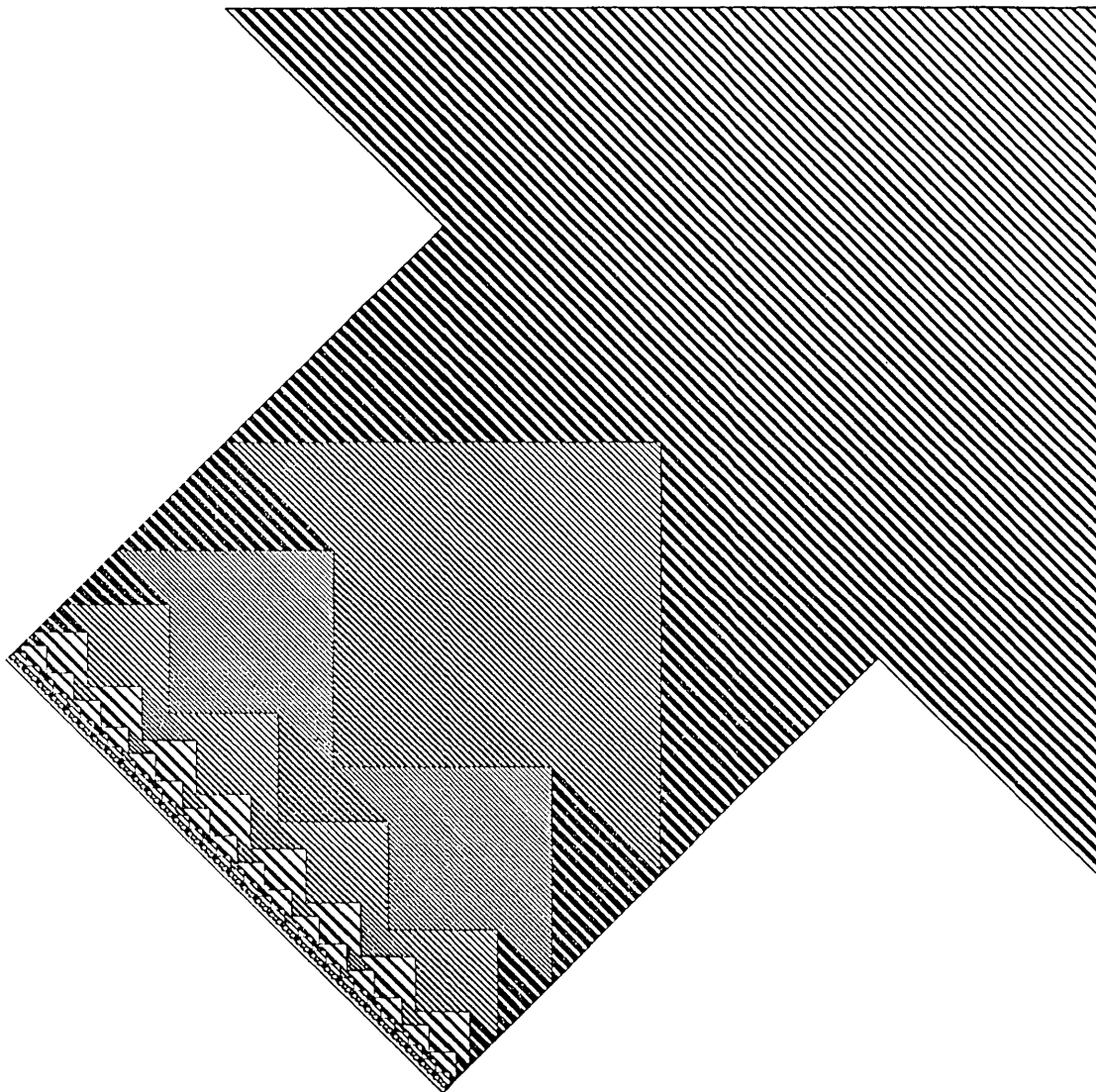
;;; Handler for bad value of *MAX-DEPTH* or *FILL-PROPORTION*.
;;; Argument is the condition object created by SIGNAL. Uses QUERY-IO
;;; stream to report condition. Sends the condition object a :PROCEED
;;; message and passes back the values it returns.
(defun bad-arrow-var-handler (cond-obj &aux bl)
  ;; Find out whether this object has the right proceed type.
  ;; If not, return nil.
  (if (send cond-obj ':proceed-type-p
            (cond ((typep cond-obj 'bad-arrow-depth) ':new-depth)
                  ((typep cond-obj 'bad-arrow-fill-proportion)
                   ':new-proportion)))
      ;; Enclose the handling operation in an UNWIND-PROTECT so that
      ;; if we use a blinker we are sure to turn it off
      (unwind-protect
       (progn
        ;; Use a blinker if the QUERY-IO stream is a window
        (setq bl (if (typep query-io 'tv:sheet)
                    ;; If a cursor-following blinker exists, use it
                    (or (tv:sheet-following-blinker query-io)
                        ;; Otherwise, make a new blinker
                        (tv:make-blinker query-io
   'tv:rectangular-blinker
   ':follow-p t))))
        ;; If a blinker, make it blink
        (if bl (send bl ':set-visibility ':blink))
        ;; Alert the user
        (tv:beep)
        ;; Send a report, presumably describing the condition
        (send cond-obj ':report query-io)
        ;; Send object a :PROCEED message and return the values
        ;; that the method returns
        (send cond-obj ':proceed
              (cond ((typep cond-obj 'bad-arrow-depth) ':new-depth)
                    ((typep cond-obj 'bad-arrow-fill-proportion)
                     ':new-proportion))))
       ;; If a blinker, turn it off
       (if bl (send bl ':set-visibility nil))))))

```

```
;;; This macro expression causes combined methods to be compiled at
;;; compile time and data structures to be generated at load time.
;;; Otherwise, these things happen at run time, when the first
;;; instance of a flavor is made.
(compile-flavor-methods arrow-window lgp-pixel-stream
                          bad-arrow-depth bad-arrow-fill-proportion)
```

## 34. Graphic Output of the Sample Program

The program used as an example in this document draws the recursive arrow graphic on the document's cover. This section contains a reproduction of the LGP graphic the program produces. For the Lisp code that calculates coordinates for the endpoints of the lines that compose the figure: See the section "Calculation Module for the Sample Program", page 383. For the code that defines the flavors and methods that mediate between the program and the system output operations: See the section "Output Module for the Sample Program", page 403.







## Index

|   |                                                                       |                                                                                                                                                                   |
|---|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| # |                                                                       | #                                                                                                                                                                 |
|   |                                                                       | #m sharp-sign reader macro 137<br>#q sharp-sign reader macro 137                                                                                                  |
| ( |                                                                       | (                                                                                                                                                                 |
|   | Start Kbd Macro (c-X                                                  | (dbg:arg) 32<br>( ) Zmacs command 292                                                                                                                             |
| ) |                                                                       | )                                                                                                                                                                 |
|   | End Kbd Macro (c-X                                                    | ) ) Zmacs command 292                                                                                                                                             |
| * |                                                                       | *                                                                                                                                                                 |
|   |                                                                       | * 24<br>*debug-io* 24<br>*error-output* 24<br>*package* 24<br>*print-base* 24<br>*query-io* 24<br>*standard-input* 24<br>*standard-output* 24<br>*terminal-io* 24 |
| + |                                                                       | +                                                                                                                                                                 |
|   |                                                                       | + 24                                                                                                                                                              |
| . |                                                                       | .                                                                                                                                                                 |
|   | Sys:site;system-name<br>Sys:site;System-name<br>Sys:site;Logical-host | .system 181<br>.System File 181, 183<br>.Translations File 182                                                                                                    |
| 1 |                                                                       | 1                                                                                                                                                                 |
|   | One Window (c-X                                                       | 1) Zmacs command 293                                                                                                                                              |
| 2 |                                                                       | 2                                                                                                                                                                 |
|   | Two Windows (c-X                                                      | 2) Zmacs command 293                                                                                                                                              |

|          |                           |                  |          |
|----------|---------------------------|------------------|----------|
| <b>3</b> |                           | <b>3</b>         | <b>3</b> |
|          | View Two Windows (c-X     | 3) Zmacs command | 293      |
| <b>4</b> |                           | <b>4</b>         | <b>4</b> |
|          | Modified Two Windows (c-X | 4) Zmacs command | 293      |
| <b>;</b> |                           | <b>;</b>         | <b>;</b> |
|          | Set Comment Column (c-X   | ;) Zmacs command | 249      |

|          |                       |                                                        |          |
|----------|-----------------------|--------------------------------------------------------|----------|
| <b>A</b> |                       | <b>A</b>                                               | <b>A</b> |
|          | Areas (               | A) 101                                                 |          |
|          |                       | ABORT 329                                              |          |
|          | Using                 | ABORT And RESUME in the Debugger 21                    |          |
|          | <b>sys:</b>           | <b>abort</b> flavor 364                                |          |
|          |                       | ABORT key 11, 12, 15, 20, 25, 50                       |          |
|          |                       | ABORT key and traps 52                                 |          |
|          |                       | :Abort command 50                                      |          |
|          |                       | Abort Command 50                                       |          |
|          |                       | Abort Patch (m-X) 209                                  |          |
|          | Disable               | Aborts Command 51                                      |          |
|          | Enable                | Aborts Command 51                                      |          |
|          | Obtaining Information | About a System 213                                     |          |
|          | Finding Out           | About Existing Code 260                                |          |
|          | Finding Out           | About Functions 265                                    |          |
|          | Finding Out           | About Objects 260                                      |          |
|          | Send mail             | about patch 208                                        |          |
|          | Finding Out           | About Pathnames 271                                    |          |
|          | Finding Out           | About Symbols 263                                      |          |
|          | Finding Out           | About Variables 265                                    |          |
|          | Debugger Command      | Accelerators 18                                        |          |
|          | Numeric arguments in  | accelerators 18                                        |          |
|          | Summary of Compiler   | Actions on Code in a Zmacs Buffer 299                  |          |
|          |                       | Active functions 41                                    |          |
|          |                       | Active patches 203, 207                                |          |
|          | Display status of     | active processes 99                                    |          |
|          | Select                | Activity command 233                                   |          |
|          | <b>compiler:</b>      | <b>add-optimizer</b> special form 135                  |          |
|          |                       | Add Patch Changed Definitions (m-X) 207                |          |
|          |                       | Add Patch Changed Definitions of Buffer (m-X) 206      |          |
|          |                       | Add Patch (m-X) 206                                    |          |
|          |                       | Add region to patch file 203                           |          |
|          |                       | : <b>advertised-in</b> Option For <b>defsystem</b> 151 |          |
|          | : <b>after</b>        | advice 82                                              |          |
|          | : <b>around</b>       | Advice 82                                              |          |
|          | : <b>before</b>       | advice 82                                              |          |
|          | Designing the         | Advice 82                                              |          |
|          |                       | Advice to functions 79                                 |          |
|          |                       | <b>advise</b> special form 79                          |          |
|          | <b>si:</b>            | <b>advise-1</b> function 81                            |          |
|          | <b>si:</b>            | <b>advised-functions</b> variable 82                   |          |
|          |                       | Advise facility 5                                      |          |
|          |                       | <b>advise-within</b> special form 83                   |          |
|          |                       | Advising a Function 79                                 |          |
|          |                       | Advising One Function Within Another 83                |          |
|          |                       | : <b>after</b> advice 82                               |          |

- [Break after] **trace** menu item 323
- [Cond after] **trace** menu item 323
- [Cond break after] **trace** menu item 323, 329
- [Print after] **trace** menu item 323
- Debugging aids 5
- Miscellaneous Debugging Aids 91
- Programming Aids for Flavors and Windows 377
- Aligning Code 251
- Clear All Breakpoints Command 59
- Select All Buffers As Tag Table (m-X) Zmacs command 284
- Macro Expand Expression All (m-sh-M) Zmacs command 332
- zl:** **all-special-switch** variable 133
- Dependencies among modules 154
- Analyze Frame Command 43
- Anonymous module 153, 156, 159
- Another 83
- Advising One Function Within
  - :any-tyl** method of **tv:** **any-tyl-mixin** 364
  - tv:** **any-tyl-mixin** flavor 364
  - applyhook** 88
  - applyhook** function 89
  - applyhook** variable 89
  - zl:** **apropos** function 263
  - Function **Apropos** (m-X) Zmacs command 266
- Display status of window area 99
- Display status of areas 99
- Areas (A) 101
- dbg:** **arg** function 64
- :arg** option for **trace** 323
- :arg** option for **trace** 76
- arglist** function 269
- arglist** variable 74
- arglist** variable 329
- Quick **Arglist** (c-sh-A) Zmacs command 269
- Show **Arglist** Command 32
- :argpdl** option for **trace** 323
- :argpdl** option to **trace** 75
- [ARGPDL]** **trace** menu item 323
- :argpdl trace** Option 75
- Show **Argument** Command 32
- Show Rest **Argument** Command 35
- Argument** Lists 269
- Ignored arguments 130
- Numeric arguments in accelerators 18
- patch-atom* argument to **:patchable** option for **defsystem** 201
- :arg :value :both nil trace** Options 76
- :around** Advice 82
- Moving around in the stack 37
- Bit-save array 349, 364
- Inspecting an array 97
- Arrays in compiled code files 139
- Mouse cursor as an arrow 95
- The Basic Arrow Window 349
- The Arrow Window: Interaction, Processes, and the Mouse 364
- Directories Associated with a System 191
- Atom Word Mode (m-X) Zmacs command 236
- Base attribute 235
- Syntax attribute 235
- Creating a File Attribute List 234
- Reparse Attribute List (m-X) Zmacs command 234

Update Attribute List (m-X) Zmacs command 234  
 Buffer attributes 234  
 [Attributes] System menu item 377  
 Auto Fill Mode (m-X) Zmacs command 236

**B****B****B**

Set Backspace (m-X) Zmacs command 234  
 Brief backtrace 41  
 Detailed backtrace 41  
 Show Backtrace Command 45  
 Backtrace information 66  
 Backtrace of the call stack 66  
 Backtraces 41, 45  
 Backward Kill Sexp (c-m-RUBOUT) Zmacs  
 command 288  
 Balancing Parentheses 252  
 Base 234  
 Base attribute 235  
 Set Base (m-X) Zmacs command 234  
 The Basic Arrow Window 349  
 :send-command method of lgp:: **basic-lgp-stream** 357  
 :send-coordinates method of lgp:: **basic-lgp-stream** 357  
 lgp:: **basic-lgp-stream** flavor 357  
 :batch option for **make-system** 309  
 Beep (c-G) Zmacs command 286  
 :before advice 82  
 [Break before] **trace** menu item 323  
 [Cond before] **trace** menu item 323  
 [Cond break before] **trace** menu item 323, 329  
 [Print before] **trace** menu item 323  
 bin file dumper 109, 117  
 bin file type 119  
 SUPER key bindings 12, 20  
 Show Bindings Command 46  
 Rebound Variable Bindings During Evaluation 24  
 Bit-save array 349, 364  
 :blinker-p init option for **tv:sheet** 349  
 Blinkers 369  
 Show Catch Blocks Command 46  
 defsystem body 153  
 :both option for **trace** 323  
 :both option for **trace** 76  
 :both nil **trace** Options 76  
 Bottom of Stack Command 38  
 Debugger break 11  
 Entering the Debugger With **break** And **zl:dbg** Functions 14  
**break** function 14  
 BREAK Inspector command 97  
 :break option for **trace** 323, 329  
 :break option to **trace** 74  
 zl: **break** special form 329  
 [Break after] **trace** menu item 323  
 [Cond break after] **trace** menu item 323, 329  
 [Break before] **trace** menu item 323  
 [Cond break before] **trace** menu item 323, 329  
 Debugger breakpoints vs. break loops 57  
 Break loops vs. Debugger breakpoints 13  
 Entering a Break Loop With SUSPEND, c-SUSPEND 13  
**breakon** function 329  
 Debugger breakpoint 13

Clear Breakpoint Command 59  
     Set Breakpoint Command 60  
 Break loops vs. Debugger breakpoints 13  
     Setting Debugger breakpoints 57  
     Using Breakpoints 329  
 Debugger Commands for Breakpoints and Single Stepping 57  
     Clear All Breakpoints Command 59  
     Show Breakpoints Command 61  
     Debugger breakpoints vs. break loops 57  
         :break trace Option 74  
         Brief backtrace 41  
         Brief Documentation (c-sh-D) Zmacs  
             command 265, 267  
         Buffer 299  
 Compiling Code in a Zmacs Buffer 299  
 Summary of Compiler Actions on Code in a Zmacs Buffer 299  
     Buffer attributes 234  
     Select Previous Buffer (c-m-L) Zmacs command 286  
     Select Buffer (c-X B) Zmacs command 286  
 Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 299  
 Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 303  
 Add Patch Changed Definitions of Buffer (m-X) 206  
     Compile Buffer (m-X) Zmacs command 299  
 Edit Changed Definitions Of Buffer (m-X) Zmacs command 282  
     Evaluate Buffer (m-X) Zmacs command 303  
 Evaluate And Replace Into Buffer (m-X) Zmacs command 303  
     Evaluate Into Buffer (m-X) Zmacs command 303  
     Insert Buffer (m-X) Zmacs command 291  
 List Changed Definitions Of Buffer (m-X) Zmacs command 282  
     Multiple Buffers 293  
     Copying Buffers and Files 291  
     Select All Buffers As Tag Table (m-X) Zmacs command 284  
     Mailing a bug report 61  
     Mail Bug Report Command 62  
         :bug-reports Option For defsystem 151  
         B) Zmacs command 286  
     Select Buffer (c-X

## C

## C

## C

c-X C 56  
     Replace ( c-%) Zmacs command 284  
 Indent For Comment ( c- ; or m- ; ) Zmacs command 249  
     c-? Zmacs minibuffer command 232  
     c-X c-A 32  
         c-B 45  
         c-B Stepper command 325  
         c-C 53  
         c-X c-D 35  
 Display Directory (c-X c-D) Zmacs command 271  
     c-E 61  
     c-X c-E 53  
         c-E Stepper command 325  
         c-F) Zmacs command 234  
 Find File (c-X c-G) Zmacs command 286  
     Beep ( c-HELP 10, 19, 63  
         c-HELP Debugger command 310  
         c-X c-I 49  
         c-L 34  
         :Show Frame ( c-L) 19  
         c-M 62  
     Kill Comment ( c-m- ; ) Zmacs command 249  
         c-m-A 32

|                                       |             |                          |
|---------------------------------------|-------------|--------------------------|
|                                       | c-m-B       | 45                       |
| c-X                                   | c-m-C       | 56                       |
|                                       | c-m-D       | 45                       |
|                                       | c-m-F       | 34                       |
| Mark Definition (                     | c-m-H)      | Zmacs command 288        |
|                                       | c-m-I       | 47                       |
| Kill Sexp (                           | c-m-K)      | Zmacs command 288        |
|                                       | c-m-L       | 34                       |
| Select Previous Buffer (              | c-m-L)      | Zmacs command 286        |
|                                       | c-m-N       | 39                       |
|                                       | c-m-P       | 40                       |
| Indent Sexp (                         | c-m-Q)      | Zmacs command 251        |
|                                       | c-m-R       | 51                       |
| Backward Kill Sexp (                  | c-m-RUBOUT) | Zmacs command 288        |
| Move To Previous Point (              | c-m-SPACE)  | Zmacs command 286        |
|                                       | c-m-SUSPEND | 310                      |
| Entering the Debugger With m-SUSPEND, | c-m-SUSPEND | 13                       |
| Indent For Lisp (TAB or               | c-m-TAB)    | Zmacs command 251        |
|                                       | c-m-U       | 40                       |
|                                       | c-m-V       | 36                       |
| Scroll Other Window (                 | c-m-V)      | Zmacs command 293        |
|                                       | c-m-W       | 63                       |
| :Window Debugger (                    | c-m-W)      | command 19               |
|                                       | c-m-Y       | input editor command 306 |
|                                       | c-m-Z       | 43                       |
| Indent Region (                       | c-m-\)      | Zmacs command 251        |
|                                       | c-N         | 39                       |
|                                       | c-N         | Stepper command 325      |
|                                       | c-P         | 40                       |
|                                       | c-R         | 51                       |
| Reverse Search (                      | c-R)        | Zmacs command 284        |
|                                       | c-S         | 38                       |
| Quick Arglist (                       | c-sh-A)     | Zmacs command 269        |
|                                       | c-sh-C)     | Zmacs command 117        |
| Compile Region (                      | c-sh-C)     | Zmacs command 299        |
| Brief Documentation (                 | c-sh-D)     | Zmacs command 265, 267   |
| Evaluate Region (                     | c-sh-E)     | Zmacs command 303        |
| Macro Expand Expression (             | c-sh-M)     | Zmacs command 332        |
|                                       | c-sh-P      | proceed option 21        |
|                                       | c-sh-S      | 61                       |
| Describe Variable At Point (          | c-sh-V)     | Zmacs command 265        |
| Set Pop Mark (                        | c-SPACE)    | Zmacs command 286        |
| Entering a Break Loop With SUSPEND,   | c-SUSPEND   | 13                       |
| Incremental Search (                  | c-S)        | Zmacs command 284        |
|                                       | c-T         | 52                       |
|                                       | c-U         | Stepper command 325      |
| Start Kbd Macro (                     | c-X ( )     | Zmacs command 292        |
| End Kbd Macro (                       | c-X )       | Zmacs command 292        |
| One Window (                          | c-X 1)      | Zmacs command 293        |
| Two Windows (                         | c-X 2)      | Zmacs command 293        |
| View Two Windows (                    | c-X 3)      | Zmacs command 293        |
| Modified Two Windows (                | c-X 4)      | Zmacs command 293        |
| Set Comment Column (                  | c-X ;)      | Zmacs command 249        |
| Select Buffer (                       | c-X B)      | Zmacs command 286        |
|                                       | c-X C       | 56                       |
|                                       | c-X c-A     | 32                       |
|                                       | c-X c-C     | 53                       |
|                                       | c-X c-D     | 35                       |
| Display Directory (                   | c-X c-D)    | Zmacs command 271        |
|                                       | c-X c-E     | 53                       |
| Find File (                           | c-X c-F)    | Zmacs command 234        |

- c-X c-I 49
- c-X c-m-C 56
- Swap Point And Mark ( c-X c-X) Zmacs command 286
- REFRESHr Dired ( c-X D) Zmacs command 271
- c-X E 56
- Call Last Kbd Macro ( c-X E) Zmacs command 292
- Set Fill Column ( c-X F) Zmacs command 236
- Open Get Register ( c-X G) Zmacs command 291
- Mark Whole ( c-X H) Zmacs command 291
- c-X I 49, 50
- Jump To Saved Position ( c-X J) Zmacs command 286
- c-X m-C 55
- Other Window ( c-X O) Zmacs command 293
- c-X Stepper command 325
- Save Position ( c-X S) Zmacs command 286
- c-X T 54
- Put Register ( c-X X) Zmacs command 291
- c-x-B 46
- Swap Point And Mark (c-X c-X) Zmacs command 286
- Yank ( c-Y) Zmacs command 288
- c-Z 50
- c-Z Inspector command 97
- Quit ( c-Z) Zmacs command 310
- Calculation Module for the Sample Program 383
- Call Command 53
- Proceed Trap on Call Command 55
- Restart Trap on Call Command 56
- Set Trap on Call Command 56
- Gallers 270
- Edit Callers (m-X) Zmacs command 270
- List Callers (m-X) Zmacs command 263, 270
- Multiple Edit Callers (m-X) Zmacs command 270
- Multiple List Callers (m-X) Zmacs command 270
- Calling the Window Debugger 61, 63
- Call Last Kbd Macro (c-X E) Zmacs command 292
- call stack 66
- Backtrace of the Capabilities 10
- Overview of Debugger Mouse :case method combination 369
- Show Catch Blocks Command 46
- Entering the Debugger by Causing an Error 11
- Caveat to Program Development Tools and Techniques 225
- Identifying Changed Code 282
- Add Patch Changed Definitions (m-X) 207
- Compile Changed Definitions (m-X) Zmacs command 299
- Edit Changed Definitions (m-X) Zmacs command 282
- Evaluate Changed Definitions (m-X) Zmacs command 303
- List Changed Definitions (m-X) Zmacs command 282
- Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 299
- Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 303
- Add Patch Changed Definitions of Buffer (m-X) 206
- Edit Changed Definitions Of Buffer (m-X) Zmacs command 282
- List Changed Definitions Of Buffer (m-X) Zmacs command 282
- Special :change-of-size-or-margins method of tv:sheet 349
- sct: Characters Recognized by the Inspector 97
- :function option for tv: check-system-patch-file-version function 217
- choose-variable-values 369





|                                                       |         |               |
|-------------------------------------------------------|---------|---------------|
| c-m-Y input editor                                    | command | 306           |
| c-N Stepper                                           | command | 325           |
| c-sh-C Zmacs                                          | command | 117           |
| c-U Stepper                                           | command | 325           |
| c-X Stepper                                           | command | 325           |
| c-Z Inspector                                         | command | 97            |
| Call Last Kbd Macro (c-X E) Zmacs                     | command | 292           |
| Clear All Breakpoints                                 | Command | 59            |
| Clear Breakpoint                                      | Command | 59            |
| Clear Trap on Call                                    | Command | 53            |
| Clear Trap on Exit                                    | Command | 53            |
| Compile Buffer (m-X) Zmacs                            | command | 299           |
| Compile Changed Definitions (m-X) Zmacs               | command | 299           |
| Compile Changed Definitions Of Buffer (m-sh-C) Zmacs  | command | 299           |
| Compile File                                          | Command | 118           |
| Compile File (m-X) Zmacs                              | command | 301           |
| Compile Region (c-sh-C) Zmacs                         | command | 299           |
| Compile Region (m-X) Zmacs                            | command | 117           |
| Compiler Warnings (m-X) Zmacs                         | command | 123, 309      |
| Compile System                                        | command | 180           |
| Compile System                                        | Command | 177           |
| COMPLETE Zmacs minibuffer                             | command | 232           |
| Deinstall Macro (m-X) Zmacs                           | command | 292           |
| Describe Flavor (m-X) Zmacs                           | command | 377           |
| Describe Last                                         | Command | 45            |
| Describe Variable At Point (c-sh-V) Zmacs             | command | 265           |
| Dired (m-X) Zmacs                                     | command | 271           |
| Disable Aborts                                        | Command | 51            |
| Disable Condition Tracing                             | Command | 54            |
| Disassemble (m-X) Zmacs                               | command | 335           |
| Display Directory (c-X c-D) Zmacs                     | command | 271           |
| Distribute Systems                                    | Command | 189           |
| Down Comment Line (m-N) Zmacs                         | command | 249           |
| Edit Callers (m-X) Zmacs                              | command | 270           |
| Edit Changed Definitions (m-X) Zmacs                  | command | 282           |
| Edit Changed Definitions Of Buffer (m-X) Zmacs        | command | 282           |
| Edit Combined Methods (m-X) Zmacs                     | command | 378           |
| Edit Compiler Warnings (m-X) Zmacs                    | command | 123, 309      |
| Edit Definition (m-.) Zmacs                           | command | 266, 377, 378 |
| Edit File Warnings (m-X) Zmacs                        | command | 123           |
| Edit Function                                         | Command | 61            |
| Editing a Debugger                                    | Command | 19            |
| Edit Methods (m-X) Zmacs                              | command | 293, 378      |
| Electric Shift Lock Mode (m-X) Zmacs                  | command | 236           |
| Enable Aborts                                         | Command | 51            |
| Enable Condition Tracing                              | Command | 54            |
| END Zmacs minibuffer                                  | command | 232           |
| End Kbd Macro (c-X ) Zmacs                            | command | 292           |
| Entering a Debugger                                   | Command | 17            |
| ESCAPE Inspector                                      | command | 97            |
| Evaluate And Replace Into Buffer (m-X) Zmacs          | command | 303           |
| Evaluate Buffer (m-X) Zmacs                           | command | 303           |
| Evaluate Changed Definitions (m-X) Zmacs              | command | 303           |
| Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs | command | 303           |
| Evaluate Into Buffer (m-X) Zmacs                      | command | 303           |
| Evaluate Minibuffer (m-ESCAPE) Zmacs                  | command | 303           |
| Evaluate Region (c-sh-E) Zmacs                        | command | 303           |
| Fill Long Comment (m-X) Zmacs                         | command | 249           |
| Find File (c-X c-F) Zmacs                             | command | 234           |

|                                                |         |          |
|------------------------------------------------|---------|----------|
| Find Frame                                     | Command | 38       |
| Find Unbalanced Parentheses (m-X) Zmacs        | command | 252      |
| Function Apropos (m-X) Zmacs                   | command | 266      |
| HELP Stepper                                   | command | 325      |
| HELP Zmacs                                     | command | 292      |
| HELP Zmacs minibuffer                          | command | 232      |
| Help (Debugger)                                | Command | 63       |
| Incremental Search (c-S) Zmacs                 | command | 284      |
| Indent For Comment (c-; or m-;) Zmacs          | command | 249      |
| Indent For Lisp (TAB or c-m-TAB) Zmacs         | command | 251      |
| Indent New Comment Line (m-LINE) Zmacs         | command | 249      |
| Indent New Line (LINE) Zmacs                   | command | 251      |
| Indent Region (c-m-\) Zmacs                    | command | 251      |
| Indent Sexp (c-m-Q) Zmacs                      | command | 251      |
| Insert Buffer (m-X) Zmacs                      | command | 291      |
| Insert File (m-X) Zmacs                        | command | 291      |
| Inspect                                        | command | 93       |
| Install Macro (m-X) Zmacs                      | command | 292      |
| Install Mouse Macro (m-X) Zmacs                | command | 292      |
| Jump To Saved Position (c-X J) Zmacs           | command | 286      |
| Kill Comment (c-m-;) Zmacs                     | command | 249      |
| Kill Sexp (c-m-K) Zmacs                        | command | 288      |
| Lisp Mode (m-X) Zmacs                          | command | 236      |
| List Callers (m-X) Zmacs                       | command | 263, 270 |
| List Changed Definitions (m-X) Zmacs           | command | 282      |
| List Changed Definitions Of Buffer (m-X) Zmacs | command | 282      |
| List Combined Methods (m-X) Zmacs              | command | 378      |
| List Matching Lines (m-X) Zmacs                | command | 284      |
| List Matching Symbols (m-X) Zmacs              | command | 263      |
| List Methods (m-X) Zmacs                       | command | 378      |
| Load Compiler Warnings (m-X) Zmacs             | command | 123, 309 |
| Load File (m-X) Zmacs                          | command | 301      |
| Load Patches                                   | Command | 211      |
| Load System                                    | command | 180      |
| Load System                                    | Command | 175      |
| Long Documentation (m-sh-D) Zmacs              | command | 265, 267 |
| m-I Debugger                                   | command | 63       |
| m-Y input editor                               | command | 306      |
| Macro Expand Expression All (m-sh-M) Zmacs     | command | 332      |
| Macro Expand Expression (c-sh-M) Zmacs         | command | 332      |
| Mail Bug Report                                | Command | 62       |
| Mark Definition (c-m-H) Zmacs                  | command | 288      |
| Mark Whole (c-X H) Zmacs                       | command | 291      |
| Modified Two Windows (c-X 4) Zmacs             | command | 293      |
| Monitor Variable                               | Command | 54       |
| Move To Previous Point (c-m-SPACE) Zmacs       | command | 286      |
| Multiple Edit Callers (m-X) Zmacs              | command | 270      |
| Multiple List Callers (m-X) Zmacs              | command | 270      |
| Name Last Kbd Macro (m-X) Zmacs                | command | 292      |
| Next Frame                                     | Command | 39       |
| One Window (c-X 1) Zmacs                       | command | 293      |
| Open Get Register (c-X G) Zmacs                | command | 291      |
| Other Window (c-X O) Zmacs                     | command | 293      |
| Previous Frame                                 | Command | 40       |
| Print Modifications (m-X) Zmacs                | command | 282      |
| Proceed                                        | Command | 51       |
| Proceed Trap on Call                           | Command | 55       |
| Push Pop Point Explicit (m-SPACE) Zmacs        | command | 286      |
| Put Register (c-X X) Zmacs                     | command | 291      |
| Query Replace (m-%) Zmacs                      | command | 284      |
| Quick Arglist (c-sh-A) Zmacs                   | command | 269      |

|                                             |         |          |
|---------------------------------------------|---------|----------|
| Quit (c-Z) Zmacs                            | command | 310      |
| REFRESHr Dired (c-X D) Zmacs                | command | 271      |
| Reinvoke                                    | Command | 51       |
| Reparse Attribute List (m-X) Zmacs          | command | 234      |
| Replace (c-%) Zmacs                         | command | 284      |
| Restart Trap on Call                        | Command | 56       |
| Return                                      | Command | 51       |
| RETURN Zmacs minibuffer                     | command | 232      |
| Reverse Search (c-R) Zmacs                  | command | 284      |
| Save Compiler Warnings                      | Command | 123      |
| Save Position (c-X S) Zmacs                 | command | 286      |
| Save Region (m-W) Zmacs                     | command | 288      |
| Scroll Other Window (c-m-V) Zmacs           | command | 293      |
| Select Activity                             | command | 233      |
| Select All Buffers As Tag Table (m-X) Zmacs | command | 284      |
| Select Buffer (c-X B) Zmacs                 | command | 286      |
| Select Previous Buffer (c-m-L) Zmacs        | command | 286      |
| Select System As Tag Table (m-X) Zmacs      | command | 284      |
| Set Backspace (m-X) Zmacs                   | command | 234      |
| Set Base (m-X) Zmacs                        | command | 234      |
| Set Breakpoint                              | Command | 60       |
| Set Comment Column (c-X ;) Zmacs            | command | 249      |
| Set Current Frame                           | Command | 41       |
| Set Fill Column (c-X F) Zmacs               | command | 236      |
| Set Fonts (m-X) Zmacs                       | command | 234      |
| Set Key (m-X) Zmacs                         | command | 292      |
| Set Lowercase (m-X) Zmacs                   | command | 234      |
| Set Nofill (m-X) Zmacs                      | command | 234      |
| Set Package (m-X) Zmacs                     | command | 234      |
| Set Patch File (m-X) Zmacs                  | command | 234      |
| Set Pop Mark (c-SPACE) Zmacs                | command | 286      |
| Set sleep time between updates Peek         | command | 99       |
| Set Stack Size                              | Command | 63       |
| Set Tab Width (m-X) Zmacs                   | command | 234      |
| Set Trap on Call                            | Command | 56       |
| Set Trap on Exit                            | Command | 56       |
| Set Vsp (m-X) Zmacs                         | command | 234      |
| Show Arglist                                | Command | 32       |
| Show Argument                               | Command | 32       |
| Show Backtrace                              | Command | 45       |
| Show Bindings                               | Command | 46       |
| Show Breakpoints                            | Command | 61       |
| Show Catch Blocks                           | Command | 46       |
| Show Compiled Code                          | Command | 33       |
| Show Compiler Warnings                      | Command | 123      |
| Show Condition Handlers                     | Command | 47       |
| Show Frame                                  | Command | 34       |
| Show Function                               | Command | 34       |
| Show Instruction                            | Command | 47       |
| Show Lexical Environment                    | Command | 48       |
| Show Local                                  | Command | 34       |
| Show Monitored Locations                    | Command | 57       |
| Show Proceed Options                        | Command | 48       |
| Show Rest Argument                          | Command | 35       |
| Show Source Code                            | Command | 35       |
| Show Special                                | Command | 48       |
| Show Stack                                  | Command | 36       |
| Show Standard Value Warnings                | Command | 49       |
| Show System Definition                      | Command | 213      |
| Show System Modifications                   | Command | 214      |
| Show System Plan                            | Command | 171, 215 |

|                                     |                                              |           |
|-------------------------------------|----------------------------------------------|-----------|
| Show Value                          | Command                                      | 36        |
| Single Step                         | Command                                      | 61        |
| Source Compare (m-X) Zmacs          | command                                      | 282       |
| Source Compare Merge (m-X) Zmacs    | command                                      | 282       |
| SPACE Stepper                       | command                                      | 325       |
| SPACE Zmacs minibuffer              | command                                      | 232       |
| Split Screen (m-X) Zmacs            | command                                      | 293       |
| Start Kbd Macro (c-X ) Zmacs        | command                                      | 292       |
| Swap Point And Mark (c-X c-X) Zmacs | command                                      | 286       |
| Symeval in Last Instance            | Command                                      | 49        |
| Tags Query Replace (m-X) Zmacs      | command                                      | 284       |
| Tags Search (m-X) Zmacs             | command                                      | 284       |
| Throw                               | Command                                      | 52        |
| Top of Stack                        | Command                                      | 41        |
| Trace (m-X) Zmacs                   | command                                      | 323, 325  |
| Two Windows (c-X 2) Zmacs           | command                                      | 293       |
| Unmonitor Variable                  | Command                                      | 57        |
| Up Comment Line (m-P) Zmacs         | command                                      | 249       |
| Update Attribute List (m-X) Zmacs   | command                                      | 234       |
| Use Dynamic Environment             | Command                                      | 49        |
| :Use Dynamic Environment            | command                                      | 22        |
| Use Lexical Environment             | Command                                      | 50        |
| :Use Lexical Environment            | command                                      | 22        |
| View Directory (m-X) Zmacs          | command                                      | 271       |
| View Two Windows (c-X 3) Zmacs      | command                                      | 293       |
| Where Is Symbol (m-X) Zmacs         | command                                      | 263       |
| Window Debugger                     | Command                                      | 63        |
| :Window Debugger (c-m-w)            | command                                      | 19        |
| Yank (c-Y) Zmacs                    | command                                      | 288       |
| Yank Pop (m-Y) Zmacs                | command                                      | 288       |
| Debugger                            | Command Accelerators                         | 18        |
| Zmacs                               | Command Completion                           | 232       |
| Debugger                            | Command Descriptions                         | 29        |
| Debugger                            | command level                                | 25        |
| Debugger                            | command prompt                               | 7, 11, 17 |
| Evaluation environment              | command prompts                              | 22        |
| Colons in Debugger                  | commands                                     | 7, 17     |
| Debugger commands vs. CP            | commands                                     | 7, 17, 29 |
| Debugger special                    | commands                                     | 310       |
| Debugger Trap                       | Commands                                     | 52        |
| Fundamental Zmacs editing           | commands                                     | 23        |
| Getting Help for Debugger           | Commands                                     | 19        |
| Inspector                           | commands                                     | 95        |
| Miscellaneous Debugger              | Commands                                     | 63        |
| Overview of Debugger                | Commands                                     | 7         |
| Summary of Debugger                 | Commands                                     | 67        |
| Debugger                            | Commands for Breakpoints and Single Stepping | 57        |
| Debugger                            | Commands for General Information Display     | 41        |
| Debugger                            | Commands for Stack Motion                    | 37        |
| Debugger                            | Commands for System Transfer                 | 61        |
| Debugger                            | Commands for Viewing a Stack Frame           | 31        |
| Debugger                            | Commands to Continue Execution               | 50        |
| Debugger                            | commands vs. CP commands                     | 7, 17, 29 |
| Debugger                            | Command table                                | 17, 29    |
| Global                              | command table                                | 7         |
| Entering a Debugger                 | Command with the Mouse                       | 19        |
| Indent For                          | Comment (c- ; or m- ; ) Zmacs command        | 249       |
| Kill                                | Comment (c-m- ; ) Zmacs command              | 249       |
| Set                                 | Comment Column (c-X ; ) Zmacs command        | 249       |
| Indent New                          | Commenting Out Code                          | 314       |
|                                     | Comment Line (m-LINE) Zmacs command          | 249       |

- Down Comment Line (m-N) Zmacs command 249
- Up Comment Line (m-P) Zmacs command 249
- Fill Long Comment (m-X) Zmacs command 249
- Comments 249
- Source Compare (m-X) Zmacs command 282
- Source Compare Merge (m-X) Zmacs command 282
- Files That Maclisp Must Compile 137
  - compile** function 107
  - compile** function 120
  - Compile Buffer (m-X) Zmacs command 299
  - Compile Changed Definitions (m-X) Zmacs command 299
  - Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 299
- Stepping through compiled code 57, 61
  - Show Compiled Code Command 33
  - Examining a Compiled Code File 98
  - Arrays in compiled code files 139
  - Compiled code objects in compiled code files 139
  - File Types of Lisp Source and Compiled Code Files 119
  - Instances in compiled code files 139
  - Lists in compiled code files 139
  - Numbers in compiled code files 139
  - Putting Data in Compiled Code Files 139
  - Symbols in compiled code files 139
  - Inspecting a compiled code object 97
  - Compiled code objects in compiled code files 139
  - Compiled function object 109
  - Compiled functions 297
  - Compile file 118
  - compile-file** function 107, 119
- compiler:** **compile-file** function 301
- Compile File Command 118
- compiler:** **compile-file-load** function 301
- Compile File (m-X) Zmacs command 301
- compile-flavor-methods** macro 369
- Function compiler 109, 116
- How to Invoke the Compiler 107
- Introduction to the Compiler 107
- Optimizer feature of the compiler 135
- Stream compiler 109
- Structure of the Compiler 109
- The Compiler 105
  - compiler:add-optimizer** special form 135
  - compiler:compile-file** function 301
  - compiler:compile-file-load** function 301
  - compiler:compiler-verbose** variable 133
  - compiler:file-declaration** function 131
  - compiler:file-declare** function 131
  - compiler:function-defined** function 130
  - compiler:function-referenced** function 131
  - compiler:functions-defined** variable 130
  - compiler:functions-referenced** variable 130
  - compiler:inhibit-style-warnings-switch** variable 133
  - compiler:load-compiler-warnings** function 124
  - compiler:make-message-obsolete** special form 129
  - compiler:make-obsolete** special form 129
  - compiler:obsolete-function-warning-switch** variable 133

- compiler:open-code-map-switch** variable 133
- compiler:style-checker** function 127
- compiler:top-level-form** property 116
- Summary of
  - Compiler Actions on Code in a Zmacs Buffer 299
  - Compile Region (c-sh-C) Zmacs command 299
  - Compile Region (m-X) Zmacs command 117
- Specifying
  - compiler environments 137
- How the Stream
  - Compiler Handles Top-level Forms 111
  - compiler-let** 111
  - Compiler Source-Level Optimizers 135
  - Compiler Style Warnings 127
  - Compiler Switches 133
  - Compiler Tools and Their Differences 117
  - Compiler variables 130
- compiler:**
  - compiler-verbose** variable 133
  - Compiler warnings 293, 299
- Controlling
  - Compiler Warnings 127
  - Save
    - Compiler Warnings Command 123
  - Show
    - Compiler Warnings Command 123
    - Compiler Warnings Database 123
  - Print
    - compiler warnings database 123
  - Update
    - compiler warnings database 123
  - Using the
    - Compiler Warnings Database 309
    - Compiler Warnings (m-X) Zmacs command 123, 309
    - Compiler Warnings (m-X) Zmacs command 123, 309
    - Compiler Warnings (m-X) Zmacs command 123, 309
  - Edit
    - :compile-satisfies-load** Option For **:module** 164
  - Load
    - compile-system** function 179
    - compile-system** Keywords 179
    - Compile System Command 177
    - Compile System command 180
    - Compile-time dependency 161
    - Compiling and Evaluating Lisp 297
    - Compiling and Loading a File 301
- Tools for
  - Compiling Code From the Editor Into Your World 117
  - Compiling Code in a Zmacs Buffer 299
- Tools for
  - Compiling Files 118
  - Compiling Lisp Code 298
- Tools for
  - Compiling Single Functions 120
- Loading and
  - Compiling Systems 175
- Zmacs Command
  - COMPLETE Zmacs minibuffer command 232
  - Completion 232
  - Complex operations 172
  - Component-directory 191
  - Component directory 149, 167, 197, 200
  - Component-directory-file 191
  - Component Directory File 193
  - Component systems 159
  - :cond** option for **trace** 323
  - :cond** option to **trace** 75
  - [Cond after] **trace** menu item 323
  - [Cond before] **trace** menu item 323
  - [Cond break after] **trace** menu item 323, 329
  - [Cond break before] **trace** menu item 323, 329
  - condition** flavor 369
  - [Conditional] **trace** menu item 323
  - condition-bind** special form 369
- Debugging
  - condition handlers 52
- Show
  - Condition Handlers Command 47
- Defining Flavors to Signal
  - Conditions 369
- Disable
  - Condition Tracing Command 54

Enable Condition Tracing Command 54  
 :cond trace Option 75  
 Introduction to the System Construction Tool 143  
 System Construction Tool 143  
 Contents of the Patch Directory Files 194  
 Debugger Commands to Continue Execution 50  
 Controlling Compiler Warnings 127  
 Controlling the Evaluation of Top-level Forms 115  
 Controlling the Format Of trace Output 77  
 Converting LGP to Screen Coordinates 354  
 Coordinates 354  
 Copying Buffers and Files 291  
 Counter Metering 219  
 Program CP commands 7, 17, 29  
 Debugger commands vs. Creating a File Attribute List 234  
 Creating a logical host 181  
 Creating a New File 234  
 Set Current Frame Command 41  
 Current patch 207  
 Current stack frame 31  
 Debugger functions to return values in current stack frame 63  
 Mouse cursor as an arrow 95

**D**

Compiler Warnings  
 Print compiler warnings  
 Update compiler warnings  
 Using the Compiler Warnings

Putting

z!:

Calling the Window  
 Editing a Form in the  
 Entering and Exiting the  
 Entering the  
 Evaluating a Form in the  
 Exiting the  
 Functions used inside the

**D**

:daemon method combination 345, 349, 354

Daemon methods 349

Database 123

database 123

database 123

Database 309

Data in Compiled Code Files 139

dbg function 14, 329

dbg:\*bound-handlers\* 24

dbg:\*default-handlers\* 24

dbg:\*show-backtrace\* variable 66

dbg:arg function 64

dbg:\*debug-io-override\* variable 66

dbg:\*defer-package-dwim\* variable 66

dbg:\*frame\* variable 66

dbg:fun function 64

dbg:loc function 64

dbg:monitor-instance-variable function 65

dbg:monitor-location function 64

dbg:\*show-backtrace\* variable 66

dbg:unmonitor-location function 65

dbg:val function 64

(dbg:fun) 34

(dbg:loc) 34

dbg:old-standard-input 24

dbg:old-standard-output 24

dbg:old-terminal-io 24

(dbg:val) 36

Debugger 3, 309, 329

Debugger 61, 63

Debugger 23

Debugger 11

Debugger 11

Debugger 22

Debugger 15

Debugger 63

**D**



- General uses of Debugger 5
- Overview of the Debugger 5
- Proceeding and Restarting in the Debugger 20
- Tools: Using the Debugger 310
- Using ABORT And RESUME in the Debugger 21
- Using the Debugger 17
- Using the Mouse in the Debugger 27
- Window Debugger 5, 310
- Debugger break 11
- Debugger breakpoint 13
- Break loops vs. Debugger breakpoints 13
- Setting Debugger breakpoints 57
- Debugger breakpoints vs. break loops 57
- Entering the Debugger by Causing an Error 11
- :Window Debugger (c-m-W) command 19
- c-HELP Debugger command 310
- Editing a Debugger Command 19
- Entering a Debugger Command 17
- Help (Debugger) Command 63
- m-I Debugger command 63
- Window Debugger Command 63
- Debugger Command Accelerators 18
- Debugger Command Descriptions 29
- Debugger command level 25
- Debugger command prompt 7, 11, 17
- Colons in Debugger commands 7, 17
- Getting Help for Debugger Commands 19
- Miscellaneous Debugger Commands 63
- Overview of Debugger Commands 7
- Summary of Debugger Commands 67
- Debugger Commands for Breakpoints and Single Stepping 57
- Debugger Commands for General Information Display 41
- Debugger Commands for Stack Motion 37
- Debugger Commands for System Transfer 61
- Debugger Commands for Viewing a Stack Frame 31
- Debugger Commands to Continue Execution 50
- Debugger commands vs. CP commands 7, 17, 29
- Entering a Debugger Command with the Mouse 19
- Debugger error 11
- Overview of Debugger Evaluation Environment 9
- Debugger Functions 63
- Debugger functions to return values in current stack frame 63
- Overview of Debugger Help Facilities 10
- Using Recursive Debugger Invocations 25
- [Edit] Window Debugger menu item 310
- [Retry] Window Debugger menu item 310
- Overview of Debugger Mouse Capabilities 10
- Mouse-sensitive Debugger output 10, 19, 27
- Debugger Proceed and Restart Options 12
- Using Debugger Proceed and Restart Options 20
- Debugger read-eval-print loop 9, 13
- Debugger special commands 310
- Debugger trap 11
- Debugger Trap Commands 52
- Debugger Variables 66
- Entering the Debugger With **break** And **zl:dbg** Functions 14
- Entering the Debugger With **m-SUSPEND**, **c-m-SUSPEND** 13
- Debugging aids 5

- Miscellaneous
  - Debugging Aids 91
  - Debugging condition handlers 52
  - Debugging Lisp Programs 309
  - Debugging tools 5
  - dbg:** **\*debug-!o-override\*** variable 66
- System
  - [DeCache] Inspector menu item 95
  - Declaration File 183, 191
  - declare** 116
  - def** 111
  - :default-init-plist** option for **defflavor** 349
  - :default-module-type** Option For **defsystem** 148
  - :default-package** Option For **defsystem** 147
  - :default-pathname** Option For **defsystem** 148
  - Defaults 29
- Mentioned
  - defaults 29
  - defconst** 111
  - zl:** **defconst** special form 298
  - defconstant** 111
  - dbg:** **\*defer-package-dwim\*** variable 66
  - deff** 111
  - defflavor** 111
  - defflavor** 349
  - defflavor** 377
  - defflavor** 345
  - defflavor** 357, 359
  - defflavor** 349
  - defflavor** 345
  - defflavor** macro 345, 349
  - sct:** **define-module-type** function 171
  - sct:** **define-system-operation** macro 172
  - Defining a System 145
  - Defining Flavors to Signal Conditions 369
- Mark
  - Definition (c-m-H) Zmacs command 288
- Show System
  - Definition Command 213
  - Definition (m-.) Zmacs command 266, 377, 378
  - Definitions 266
  - :definitions** clause 156
- Edit
  - Definitions (m-X) 207
  - Definitions (m-X) Zmacs command 299
  - Definitions (m-X) Zmacs command 282
  - Definitions (m-X) Zmacs command 303
  - Definitions (m-X) Zmacs command 282
  - Definitions Of Buffer (m-sh-C) Zmacs command 299
  - Definitions Of Buffer (m-sh-E) Zmacs command 303
  - Definitions of Buffer (m-X) 206
  - Definitions Of Buffer (m-X) Zmacs command 282
  - Definitions Of Buffer (m-X) Zmacs command 282
  - Definitions of functions 137
  - Definitions That Use Logical Pathnames 180
  - Definitions That Use Physical Pathnames 185
- defmacro** 111
  - defmethod** 111
  - defpackage** 111
  - defselect** 111
  - defstruct** 111
  - defsubst** 111
  - defsubsystem** special form 167
  - defsystem** 111
  - defsystem** 151
  - defsystem** 151
  - defsystem** 148
- :advertised-in** Option For
  - :bug-reports** Option For
  - :default-module-type** Option For
- Add Patch Changed
  - Compile Changed
  - Edit Changed
  - Evaluate Changed
  - List Changed
- Compile Changed
  - Definitions Of Buffer (m-sh-C) Zmacs command 299
- Evaluate Changed
  - Definitions Of Buffer (m-sh-E) Zmacs command 303
- Add Patch Changed
  - Edit Changed
  - List Changed
- Loading System
  - Definitions That Use Logical Pathnames 180
  - Definitions That Use Physical Pathnames 185
- Loading System
  - Definitions That Use Logical Pathnames 180
  - Definitions That Use Physical Pathnames 185

- :default-package** Option For **defsystem** 147
- :default-pathname** Option For **defsystem** 148
- :distribute-binaries** Option For **defsystem** 152
- :distribute-sources** Option For **defsystem** 152
- :initializations** Option For **defsystem** 150
- :initial-status** Option For **defsystem** 151
- :journal-directory** Option For **defsystem** 149
- :maintaining-sites** Option For **defsystem** 152
- :module** option for **defsystem** 158
- :package-override** Option For **defsystem** 147
- :parameters** Option For **defsystem** 150
- :patchable** Option For **defsystem** 149, 197, 200
- patch-atom* argument to **:patchable** option for **defsystem** 201
- :pathname-default** option for **defsystem** 197, 200
- :pretty-name** Option For **defsystem** 146
- :short-name** Option For **defsystem** 146
- :source-category** Option For **defsystem** 152
- defsystem** body 153
- defsystem** Modules 153
- defsystem** Operations 168
- defsystem** Options 146
- defsystem** special form 146, 197, 284
- defun** 111
- defvar** 111
- defvar** special form 240, 298
- defvar-standard** 111
- zl-user:** **defwindow-resource** special form 359
- Deinstall Macro (m-X) Zmacs command 292
- Density and Spacing 272
- Dependencies 154
- Dependencies among modules 154
- Dependency 153
- Compile-time dependency 161
- Load-time dependency 161
- Deriving Methods for Tools and Techniques 227
- describe** function 45
- describe** function 260, 335
- Described in Tools and Techniques 228
- zl-user:** **describe-flavor** function 377
- Describe Flavor (m-X) Zmacs command 377
- Describe Last Command 45
- describe-system** function 213
- Describe Variable At Point (c-sh-V) Zmacs command 265
- Debugger Command Descriptions 29
- Program Development: Design and Figure Outline 238
- sct:** **designate-system-version** function 189
- Designing the Advice 82
- Detailed backtrace 41
- Program Development: Design and Figure Outline 238
- Program Development: Drawing Stripes 252
- Program Development: Modifying the Output Module 344
- Program Development: Refining Stripe Density and Spacing 272
- Caveat to Program Development Tools and Techniques 225
- Program Development Tools and Techniques 225
- Introduction to Program Development Utilities 1
- Compiler Tools and Their Differences 117
- Directories 271
- Directories Associated with a System 191
- Component directory 149, 167, 197, 200

Journal directory 149, 191  
 Patch directory 149, 197, 200  
 System directory 149  
 Display Directory (c-X c-D) Zmacs command 271  
 Component Directory File 193  
 File types of the patch directory file 201  
 Patch Directory File 200  
 Contents of the Patch Directory Files 194  
 View Directory (m-X) Zmacs command 271  
 REFRESHr Dired (c-X D) Zmacs command 271  
 Dired (m-X) Zmacs command 271  
 Disable Aborts Command 51  
 Disable Condition Tracing Command 54  
**disassemble** function 335  
 Disassemble (m-X) Zmacs command 335  
 Display 41  
 display 99  
 Display 11  
 Display 97  
 Display Directory (c-X c-D) Zmacs command 271  
 Displaying Zmacs and Other Windows 294  
 Displays 295  
 Display status of active processes 99  
 Display status of areas 99  
 Display status of file system display 99  
 Display status of hostat 99  
 Display status of window area 99  
 Display system information 99  
**:distribute-binaries** Option For **defsystem** 152  
**:distribute-binaries** Option For **:module** 166  
**:distribute-sources** Option For **defsystem** 152  
**:distribute-sources** Option For **:module** 166  
 Distribute Systems Command 189  
 Distribution dumper 152  
**documentation** function 265, 267  
**:documentation** option for **deffavor** 377  
 Brief Documentation (c-sh-D) Zmacs command 265, 267  
 Long Documentation (m-sh-D) Zmacs command 265, 267  
 Mouse documentation string 364  
 Documentation strings 265, 267  
 Down Comment Line (m-N) Zmacs command 249  
 Drawing Stripes 252  
**:draw-line** method of **tv:graphics-mixin** 239, 349  
 dumper 109, 117  
 dumper 152  
**bin** file  
 Distribution  
**sys:** **dump-forms-to-file** function 139  
 Rebound Variable Bindings During Evaluation 24  
 Use Dynamic Environment Command 49  
 :Use Dynamic Environment command 22  
 Dynamic evaluation environment 22  
 REFRESHr Dired (c-X D) Zmacs command 271

## E

## E

## E

c-X E 56  
 SELECT E 233  
 Multiple  
**:edges-from** init option for **tv:essential-window** 349  
 Edit Callers (m-X) Zmacs command 270  
 Edit Callers (m-X) Zmacs command 270  
 Edit Changed Definitions (m-X) Zmacs command 282  
 Edit Changed Definitions Of Buffer (m-X) Zmacs

- command 282
- Edit Combined Methods (m-X) Zmacs command 378
- Edit Compiler Warnings (m-X) Zmacs command 123, 309
- Edit Definition (m-. ) Zmacs command 266, 377, 378
- Edit File Warnings (m-X) Zmacs command 123
- Edit Function Command 61
- Lisp Input
  - Editing 306
  - Editing a Debugger Command 19
  - Editing a Form in the Debugger 23
  - Editing a function 61
  - Editing Code 282
- Writing and Fundamental Zmacs
  - Editing Code 231
  - editing commands 23
  - Editing, Hardcopying, Reap-Protecting, and Releasing Systems 187
  - Edit Methods (m-X) Zmacs command 293, 378
- Entering the Evaluation and the Input
  - Editor 233
  - Editor 303
  - Editor 19, 23
  - c-m-Y input editor command 306
  - m-Y input editor command 306
- Tools for Compiling Code From the
  - Editor Into Your World 117
  - [Edit Screen] System menu item 294, 364
  - sct:
    - edit-system function 187
    - [Edit] System menu item 233
    - [Edit] Window Debugger menu item 310
  - Electric Shift Lock Mode (m-X) Zmacs command 236
  - Enable Aborts Command 51
  - Enable Condition Tracing Command 54
  - END Zmacs minibuffer command 232
  - End Kbd Macro (c-X ) Zmacs command 292
  - Entering a Break Loop With SUSPEND, c-SUSPEND 13
  - Entering a Debugger Command 17
  - Entering a Debugger Command with the Mouse 19
  - Entering and Exiting the Debugger 11
  - Entering and Leaving the Inspector 93
  - Entering the Debugger 11
  - Entering the Debugger by Causing an Error 11
  - Entering the Debugger With **break** And **zl:dbg** Functions 14
  - Entering the Debugger With m-SUSPEND, c-m-SUSPEND 13
  - Entering the Editor 233
  - :entry option for **trace** 323
  - :entry option to **trace** 76
  - :entrycond option for **trace** 323
  - :entrycond option to **trace** 75
  - :entrycond **trace** Option 75
  - :entryprint option for **trace** 323
  - :entryprint option to **trace** 75
  - :entryprint **trace** Option 75
  - :entry **trace** Option 76
- Dynamic evaluation
  - environment 22
- Lexical evaluation
  - environment 9, 22
- Overview of Debugger Evaluation
  - Environment 9
  - Environment Command 48
  - Environment Command 49
  - :Use Dynamic Environment command 22
  - Use Lexical Environment Command 50

- :Use Lexical Evaluation
- Specifying compiler Debugger
- Entering the Debugger by Causing an Environment command 22
- environment command prompts 22
- environments 137
- error 11
- Error 11
- error** flavor 369
- :error option for **trace** 323, 329
- :error option to **trace** 74
- Error Display 11
- error-restart-loop** special form 364
- [Error] **trace** menu item 323, 329
- :error **trace** Option 74
- ESCAPE Inspector command 97
- essential-window** 349
- essential-window** 349
- essential-window** 349
- essential-window** 349
- evalhook** 24
- evalhook** 87
- evalhook** function 87
- evalhook** variable 87
- Evalhook facility 5
- Evaluate And Replace Into Buffer (m-X) Zmacs command 303
- Evaluate Buffer (m-X) Zmacs command 303
- Evaluate Changed Definitions (m-X) Zmacs command 303
- Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 303
- Evaluate Into Buffer (m-X) Zmacs command 303
- Evaluate Minibuffer (m-ESCAPE) Zmacs command 303
- Evaluate Region (c-sh-E) Zmacs command 303
- Evaluating a Form in the Debugger 22
- Evaluating code 303, 325
- Evaluating Lisp 297
- Evaluating Lisp Code 303
- Evaluation 24
- Evaluation 85
- Evaluation and the Editor 303
- evaluation environment 22
- evaluation environment 9, 22
- Evaluation Environment 9
- Evaluation environment command prompts 22
- Evaluation of Top-level Forms 115
- eval-when** 111
- eval-when** special form 115
- Examining a Compiled Code File 98
- Examining values of instance variables 63
- Execution 50
- Execution 73
- Execution 78
- Existing Code 260
- :exit option for **trace** 76
- :exit option for **trace** 323
- :exitbreak option for **trace** 323, 329
- :exitbreak option to **trace** 74
- :exitbreak **trace** Option 74
- Exit Command 53
- Exit Command 56
- :exitcond option for **trace** 323
- :edges-from init option for **tv**:
- :expose-p init option for **tv**:
- :minimum-height init option for **tv**:
- :minimum-width init option for **tv**:
- Compiling and Rebound Variable Bindings During Stepping Through an
- Dynamic Lexical Overview of Debugger
- Controlling the Debugger Commands to Continue Tracing Function Untracing Function Finding Out About
- Clear Trap on Set Trap on

:exitcond option to trace 75  
 :exitcond trace Option 75  
 Exiting the Debugger 15  
 Entering and Exiting the Debugger 11  
 Exiting the Inspector 93  
 [Exit] Inspector menu item 95, 335  
 :exitprint option for trace 323  
 :exitprint option to trace 76  
 :exitprint trace Option 76  
 :exit trace Option 76  
 Macro Expand Expression All (m-sh-M) Zmacs  
 command 332  
 Macro Expand Expression (c-sh-M) Zmacs command 332  
 Expanding Macros 332  
**meter:** expand-range function 222  
 Push Pop Point Explicit (m-SPACE) Zmacs command 286  
 :expose-p init option for tv:essential-window 349  
**zl:** \*expr special form 130  
 Macro Expand Expression All (m-sh-M) Zmacs command 332  
 Macro Expand Expression (c-sh-M) Zmacs command 332  
 Call Last Kbd Macro (c-X E) Zmacs command 292

## F

## F

## F

File System ( F) 102  
 Overview of Debugger Help Facilities 10  
 Advise facility 5  
 Evalhook facility 5  
 Patch Facility 197  
 Step facility 5  
 System facility 143  
 Trace facility 5  
 Monitor facility functions 63  
 :fasd-form message 139  
 Fasdump 139  
 Optimizer feature of the compiler 135  
 Features Described in Tools and Techniques 228  
**zl:** \*fexpr special form 131  
 A Mixin to Position the Figure 345  
 Outlining the Figure 240  
 Program Development: Design and Figure Outline 238  
 Add region to patch file 203  
 Compile file 118  
 Compiling and Loading a File 301  
 Component Directory File 193  
 Creating a New File 234  
 Examining a Compiled Code File 98  
 File types of the patch directory file 201  
 File types of the system version-directory file 201  
 Install patch file 208  
 Patch file 191, 197  
 Patch Directory File 200  
 Sys:site;Logical-host.Translations File 182  
 Sys:site;System-name.System File 181, 183  
 System file 181  
 System Declaration File 183, 191  
 Translations file 182  
 Creating a File Attribute List 234  
 Find File (c-X c-F) Zmacs command 234  
 Compile File Command 118  
**compiler:** file-declaration function 131

- compiler:** **file-declare** function 131
- bin** file dumper 109, 117
- sys:** **file-local-declarations** variable 130
- Compile File (m-X) Zmacs command 301
- Insert File (m-X) Zmacs command 291
- Load File (m-X) Zmacs command 301
- Set Patch File (m-X) Zmacs command 234
- Format of patch file names 201
- Arrays in compiled code files 139
- Compiled code objects in compiled code files 139
- Contents of the Patch Directory Files 194
- Copying Buffers and Files 291
- File Types of Lisp Source and Compiled Code Files 119
- Individual Patch Files 200
- Init files 236, 292
- Instances in compiled code files 139
- Journal files 149
- Lists in compiled code files 139
- Names of Patch Files 201
- Numbers in compiled code files 139
- Organization of Patch Files 200
- Putting Data in Compiled Code Files 139
- Symbols in compiled code files 139
- System source files 197
- Tools for Compiling Files 118
- Types of Patch Files 199
- Files That Maclisp Must Compile 137
- Display status of file system display 99
- File System (F) 102
- bin** file type 119
- File Types of Lisp Source and Compiled Code Files 119
- File types of the patch directory file 201
- File types of the system version-directory file 201
- Edit File Warnings (m-X) Zmacs command 123
- Set Fill Column (c-X F) Zmacs command 236
- Fill Long Comment (m-X) Zmacs command 249
- Auto Fill Mode (m-X) Zmacs command 236
- Find File (c-X c-F) Zmacs command 234
- Find Frame Command 38
- Finding Out About Existing Code 260
- Finding Out About Functions 265
- Finding Out About Objects 260
- Finding Out About Pathnames 271
- Finding Out About Symbols 263
- Finding Out About Variables 265
- Find Unbalanced Parentheses (m-X) Zmacs command 252
- Finish Patch (m-X) 208
- condition** flavor 369
- error** flavor 369
- lgp::basic-lgp-stream** flavor 357
- sl:vanilla-flavor** flavor 378
- sys:abort** flavor 364
- tv:any-tyl-mixin** flavor 364
- tv:graphics-mixin** flavor 349
- tv:list-mouse-buttons-mixin** flavor 364
- tv:process-mixin** flavor 364
- tv:sheet** flavor 349, 369
- tv>window** flavor 344, 349
- si:** **flavor-allowed-init-keywords** function 381



- Describe Flavor (m-X) Zmacs command 377
- Flavors 381
- General Information on Flavors 377
- Programming Aids for Flavors and Windows 377
- Using Flavors and Windows 343
- Flavors for LGP Output 357
- Defining Flavors to Signal Conditions 369
- Set Fonts (m-X) Zmacs command 234
- advise** special form 79
- advise-within** special form 83
- compiler:add-optimizer** special form 135
- compiler:make-message-obsolete** special form 129
- compiler:make-obsolete** special form 129
- condition-bind** special form 369
- defsubsystem** special form 167
- defsystem** special form 146, 197, 284
- defvar** special form 240, 298
- error-restart-loop** special form 364
- eval-when** special form 115
- special** special form 114
- trace** special form 73, 323, 325, 329
- unadvise** special form 81
- unadvise-within** special form 84
- untrace** special form 78, 323
- unwind-protect** special form 369
- with-open-stream** special form 359
- zl:break** special form 329
- zl:defconst** special form 298
- zl:\*expr** special form 130
- zl:\*fexpr** special form 131
- zl:\*lexpr** special form 130
- zl:multiple-value** special form 349
- zl:unspecial** special form 115
- zl-user:defwindow-resource** special form 359
- Format of patch file names 201
- Controlling the Format Of **trace** Output 77
- Editing a Form in the Debugger 23
- Evaluating a Form in the Debugger 22
- Controlling the Evaluation of Top-level Forms 115
- How the Stream Compiler Handles Top-level Forms 111
- Current stack frame 31
- Debugger Commands for Viewing a Stack Frame 31
- Debugger functions to return values in current stack frame 63
- Inspecting a stack frame 97
- dbg:** **\*frame\*** variable 66
- :Show** Frame (c-L) 19
- Analyze Frame Command 43
- Find Frame Command 38
- Next Frame Command 39
- Previous Frame Command 40
- Set Current Frame Command 41
- Show Frame Command 34
- Tools for Compiling Code From the Editor Into Your World 117
- fs:make-logical-pathname-host** 181
- fs:set-logical-pathname-host** 182
- fun** function 64
- dbg:** Advising a Function 79
- applyhook** function 89
- arglist** function 269
- break** function 14
- breakon** function 329

|                                                 |          |          |
|-------------------------------------------------|----------|----------|
| compile                                         | function | 120      |
| compile                                         | function | 107      |
| compile-file                                    | function | 107, 119 |
| compiler:compile-file                           | function | 301      |
| compiler:compile-file-load                      | function | 301      |
| compiler:file-declaration                       | function | 131      |
| compiler:file-declare                           | function | 131      |
| compiler:function-defined                       | function | 130      |
| compiler:function-referenced                    | function | 131      |
| compiler:load-compiler-warnings                 | function | 124      |
| compiler:style-checker                          | function | 127      |
| compile-system                                  | function | 179      |
| dbg:arg                                         | function | 64       |
| dbg:fun                                         | function | 64       |
| dbg:loc                                         | function | 64       |
| dbg:monitor-instance-variable                   | function | 65       |
| dbg:monitor-location                            | function | 64       |
| dbg:unmonitor-location                          | function | 65       |
| dbg:val                                         | function | 64       |
| describe                                        | function | 260, 335 |
| describe                                        | function | 45       |
| describe-system                                 | function | 213      |
| disassemble                                     | function | 335      |
| documentation                                   | function | 265, 267 |
| Editing a                                       | function | 61       |
| evalhook                                        | function | 87       |
| get-handler-for                                 | function | 378      |
| inspect                                         | function | 93, 335  |
| load-patches                                    | function | 211      |
| load-system                                     | function | 176      |
| make-system                                     | function | 309      |
| meter:expand-range                              | function | 222      |
| meter:function-name-with-escapes                | function | 223      |
| meter:function-range                            | function | 223      |
| meter:list-functions-in-bucket                  | function | 223      |
| meter:make-pc-array                             | function | 221      |
| meter:map-over-functions-in-bucket              | function | 223      |
| meter:monitor-all-functions                     | function | 222      |
| meter:monitor-between-functions                 | function | 222      |
| meter:print-functions-in-bucket                 | function | 223      |
| meter:range-of-bucket                           | function | 223      |
| meter:report                                    | function | 222      |
| meter:setup-monitor                             | function | 222      |
| meter:start-monitor                             | function | 222      |
| meter:stop-monitor                              | function | 222      |
| mexp                                            | function | 332      |
| note-private-patch                              | function | 210      |
| print-compiler-warnings                         | function | 124      |
| prompt-and-read                                 | function | 369      |
| sct:check-system-patch-file-version             | function | 217      |
| sct:define-module-type                          | function | 171      |
| sct:designate-system-version                    | function | 189      |
| sct:edit-system                                 | function | 187      |
| sct:get-all-system-input-files                  | function | 217      |
| sct:get-release-version                         | function | 216      |
| sct:get-system-input-and-output-defsystem-files | function | 216      |
| sct:get-system-input-and-output-source-files    | function | 216      |
| sct:get-system-version                          | function | 215      |
| sct:hardcopy-system                             | function | 187      |
| sct:patch-loaded-p                              | function | 216      |
| sct:patch-system-pathname                       | function | 202      |

**sct:reap-protect-system** function 188  
**sct:release-system** function 189  
**sct:set-system-source-file** function 181  
**sct:set-system-status** function 188  
**sct:system-version-info** function 215  
**si:advise-1** function 81  
**si:flavor-allowed-init-keywords** function 381  
**si:make-hardcopy-stream** function 359  
**si:unadvise-1** function 81  
**si:unbin-file** function 98  
**signal** function 344, 369  
**sys:dump-forms-to-file** function 139  
**The Top-Level Function** 359  
**throw** function 52  
**tv:choose-variable-values** function 359, 369  
**tv:make-blinker** function 369  
**tv:make-window** function 344, 354, 359, 364  
**tv:sheet-following-blinker** function 369  
**unbreakon** function 329  
**uncompile** function 120  
**what-files-call** function 263  
**where-is** function 263  
**who-calls** function 263  
**zl:apropos** function 263  
**zl:dbg** function 14, 329  
**zl:listarray** function 260  
**zl:load** function 301  
**zl:load-and-save-patches** function 212  
**zl:peek** function 99  
**zl:pkg-goto** function 240  
**zl:plist** function 263  
**zl:print-system-modifications** function 216  
**zl:step** function 85, 325  
**zl:typep** function 377  
**zl-user:describe-flavor** function 377  
**zl-user:undefsystem** function 166  
**:function** option for **tv:choose-variable-values** 369  
**Function Apropos (m-X) Zmacs command** 266  
**Edit** Function Command 61  
**Show** Function Command 34  
**Function compiler** 109, 116  
**compiler: function-defined** function 130  
**Tracing** Function Execution 73  
**Untracing** Function Execution 78  
**meter: function-name-with-escapes** function 223  
**Compiled** function object 109  
**meter: function-range** function 223  
**compiler: function-referenced** function 131  
**Function-referenced-but-never-defined** Warnings 130  
**Active** functions 41  
**Advice to** functions 79  
**Compiled** functions 297  
**Debugger** Functions 63  
**Definitions of** functions 137  
**Entering the Debugger With break And zl:dbg** Functions 14  
**Finding Out About** Functions 265  
**Interpreted** functions 297  
**Monitor facility** functions 63  
**Tools for Compiling Single** Functions 120  
**compiler: functions-defined** variable 130  
**compiler: functions-referenced** variable 130

Debugger functions to return values in current stack frame 63  
 Functions used inside the Debugger 63  
 Advising One Function Within Another 83  
 Fundamental Zmacs editing commands 23  
 Set Fill Column (c-X F) Zmacs command 236

## G

## G

## G

Debugger Commands for  
 General Information Display 41  
 General Information on Flavors 377  
 General uses of Debugger 5  
 Generic operations 344  
**sct:** **get-all-system-input-files** function 217  
**get-handler-for** function 378  
 Open Get Register (c-X G) Zmacs command 291  
**sct:** **get-release-version** function 216  
**sct:** **get-system-input-and-output-defsystem-files**  
 function 216  
**sct:** **get-system-input-and-output-source-files**  
 function 216  
**sct:** **get-system-version** function 215  
**:gettable-instance-variables** option for  
**defflavor** 345  
 Getting Help for Debugger Commands 19  
 Global command table 7  
 Graphic Output of the Sample Program 425  
**:draw-line** method of **tv:** **graphics-mixin** 239, 349  
**tv:** **graphics-mixin** flavor 349  
 Module **group** 156  
 Open Get Register (c-X G) Zmacs command 291

## H

## H

## H

Hostat ( H) 103  
 Debugging condition handlers 52  
 Proceed handlers 12, 20  
 Restart handlers 12, 20, 310  
 Show Condition Handlers Command 47  
 How the Stream Compiler Handles Top-level Forms 111  
 Editing, Hardcopying, Reap-Protecting, and Releasing  
 Systems 187  
**sct:** **hardcopy-system** function 187  
 HELP key 19  
 Using The HELP Key in Zmacs 231  
 HELP Stepper command 325  
 HELP Zmacs command 292  
 HELP Zmacs minibuffer command 232  
 Help and Quit 103  
 Help (Debugger) Command 63  
 Overview of Debugger Help Facilities 10  
 Getting Help for Debugger Commands 19  
 Help for keywords 19  
 Help Message 99  
 Peek History Pane 95  
 The Inspector host 181  
 Creating a logical hostat 99  
 Display status of Hostat (H) 103  
 How the Inspector Works 93  
 How the Stream Compiler Handles Top-level  
 Forms 111

- Mark Whole (c-X)      How to Invoke the Compiler 107
  - H) Zmacs command 291
- 
- c-X      I 49, 50
  - SELECT    I 93, 335
  - Identifying Changed Code 282
  - zl:** **if-for-lispm** macro 137
  - zl:** **if-for-maclisp** macro 137
  - zl:** **if-for-maclisp-else-lispm** macro 137
  - zl:** **if-in-lispm** macro 137
  - zl:** **if-in-maclisp** macro 138
  - ignore** variable 130
  - Ignored arguments 130
  - Inactive patches 207
  - Incremental Search (c-S) Zmacs command 284
  - Indent For Comment (c-; or m-;) Zmacs command 249
  - Indent For Lisp (TAB or c-m-TAB) Zmacs command 251
  - Indent New Comment Line (m-LINE) Zmacs command 249
  - Indent New Line (LINE) Zmacs command 251
  - Indent Region (c-m-\) Zmacs command 251
  - Indent Sexp (c-m-Q) Zmacs command 251
  - Individual Patch Files 200
  - Backtrace information 66
  - Display system information 99
  - Obtaining Information About a System 213
  - Debugger Commands for General Information Display 41
  - General Information on Flavors 377
  - Obtaining Information on System Versions 215
  - inhibit-style-warnings** macro 128
  - compiler:** **inhibit-style-warnings-switch** variable 133
  - :init** method of **tv:sheet** 349
  - :initable-instance-variables** option for **defflavor** 357, 359
  - Init files 236, 292
  - :initializations** Option For **defsystem** 150
  - Initial patch state 207
  - :initial-status** Option For **defsystem** 151
  - Init Keywords 381
  - :edges-from** init option for **tv:essential-window** 349
  - :expose-p** init option for **tv:essential-window** 349
  - :minimum-height** init option for **tv:essential-window** 349
  - :minimum-width** init option for **tv:essential-window** 349
  - :process** init option for **tv:process-mixin** 364
  - :blinker-p** init option for **tv:sheet** 349
  - :in-order-to** Option For **:module** 161
  - In-progress patch 207
  - In-progress patch state 207
  - Lisp Input Editing 306
  - Input Editor 19, 23
  - c-m-Y input editor command 306
  - m-Y input editor command 306
  - Insert Buffer (m-X) Zmacs command 291
  - Insert File (m-X) Zmacs command 291
  - :inside-size** method of **tv:sheet** 349
  - Functions used inside the Debugger 63
  - inspect** function 93, 335

- Inspect command 93
- Inspecting a closure 97
- Inspecting a compiled code object 97
- Inspecting a list 97
- Inspecting a named structure 97
- Inspecting an array 97
- Inspecting an instance 97
- Inspecting a select method 97
- Inspecting a stack frame 97
- Inspecting a symbol 97
- Inspecting objects 96
- [Inspect] in System menu 93
- Inspection Pane 96
- Inspection Pane Display 97
- Inspector 5, 91, 377
- Inspector 93
- Inspector 93
- Inspector 97
- Inspector 93
- Inspector 335
- Inspector 95, 96
- Inspector command 97
- Inspector command 97
- Inspector command 97
- Inspector commands 95
- Inspector History Pane 95
- Inspector Inspection Pane 96
- Inspector Interaction Pane 95
- Inspector menu item 95
- Inspector menu item 95
- Inspector menu item 95, 335
- Inspector menu item 95, 335
- Inspector menu item 95, 335
- Inspector menu item 95
- Inspector Menu Pane 95
- Inspector Works 93
- [Inspect] System menu item 335
- Install Macro (m-X) Zmacs command 292
- Install Mouse Macro (m-X) Zmacs command 292
- Install patch file 208
- instance 97
- Instance Command 49
- Instances in compiled code files 139
- Instance variables 345, 357, 359
- instance variables 63
- Instruction Command 47
- Interaction Pane 95
- Interaction, Processes, and the Mouse 364
- Interpreted functions 297
- Introduction to Program Development Utilities 1
- Introduction to the Compiler 107
- Introduction to the System Construction Tool 143
- Introduction to Tools and Techniques 227
- Invocations 25
- Invoke the Compiler 107
- Is Symbol (m-X) Zmacs command 263
- Inspecting an  
Symeval in Last
- Examining values of  
Show
- The Inspector
- The Arrow Window:
- Using Recursive Debugger  
How to  
Where

## J

Journal directory 149, 191  
**:journal-directory** Option For **defsystem** 149  
 Journal files 149  
 Journal subdirectory 191, 197  
 Jump To Saved Position (c-X J) Zmacs  
 command 286  
 Jump To Saved Position (c-X J) Zmacs command 286

## J

## J

## K

Start Kbd Macro (c-X ()) Zmacs command 292  
 End Kbd Macro (c-X ) Zmacs command 292  
 Call Last Kbd Macro (c-X E) Zmacs command 292  
 Name Last Kbd Macro (m-X) Zmacs command 292  
 Keeping Track of Lisp Syntax 247  
 key 11, 12, 15, 20, 25, 50  
 ABORT key 19  
 HELP key 39  
 LINE key 39  
 REFRESH key 19, 34  
 RESUME key 12, 20, 51  
 RETURN key 40  
 ABORT key and traps 52  
 RESUME key and traps 52  
 SUPER key bindings 12, 20  
 Keyboard Macros 292  
 Using The HELP Key in Zmacs 231  
 Set Key (m-X) Zmacs command 292  
**:module** Keyword Options 159  
**compile-system** Keywords 179  
 Help for keywords 19  
 Init Keywords 381  
**load-system** Keywords 176  
 Kill Comment (c-m-;) Zmacs command 249  
 Killing and Yanking 288  
 Kill Sexp (c-m-K) Zmacs command 288  
 Backward Kill Sexp (c-m-RUBOUT) Zmacs command 288

## K

## K

## L

Maintaining Large Programs 141  
 Describe Last Command 45  
 Symeval in Last Instance Command 49  
 Call Last Kbd Macro (c-X E) Zmacs command 292  
 Name Last Kbd Macro (m-X) Zmacs command 292  
 Entering and Leaving the Inspector 93  
 Debugger command level 25  
 Patch level 216  
 Show Lexical Environment Command 48  
 Use Lexical Environment Command 50  
 :Use Lexical Environment command 22  
 Lexical evaluation environment 9, 22  
**zl:** \*lexpr special form 130  
**:send-command** method of **lgp::basic-lgp-stream** 357  
**:send-coordinates** method of **lgp::basic-lgp-stream** 357  
**lgp::basic-lgp-stream** flavor 357  
 Flavors for LGP Output 357  
 Converting LGP to Screen Coordinates 354  
 LINE key 39

## L

## L

- Indent New Line (LINE) Zmacs command 251
- Indent New Comment Line (m-LINE) Zmacs command 249
- Down Comment Line (m-N) Zmacs command 249
- Up Comment Line (m-P) Zmacs command 249
- Line region 95
- Lines (m-X) Zmacs command 284
- List Matching (LINE) Zmacs command 251
- Indent New Line (LINE) Zmacs command 251
- Compiling and Evaluating Lisp 297
  - Compiling Lisp Code 298
  - Evaluating Lisp Code 303
  - Lisp Input Editing 306
  - Lisp Mode (m-X) Zmacs command 236
- Debugging Lisp Programs 309
  - Lisp read-eval-print loop 13
  - Lisp Source and Compiled Code Files 119
- File Types of Keeping Track of Lisp Syntax 247
  - Indent For Lisp (TAB or c-m-TAB) Zmacs command 251
- Creating a File Attribute List 234
  - Inspecting a list 97
  - zl: **larray** function 260
  - List Callers (m-X) Zmacs command 263, 270
  - Multiple List Callers (m-X) Zmacs command 270
  - List Changed Definitions (m-X) Zmacs command 282
  - List Changed Definitions Of Buffer (m-X) Zmacs command 282
  - List Combined Methods (m-X) Zmacs command 378
  - meter: **list-functions-in-bucket** function 223
  - Reparse Attribute List (m-X) Zmacs command 234
  - Update Attribute List (m-X) Zmacs command 234
  - List Matching Lines (m-X) Zmacs command 284
  - List Matching Symbols (m-X) Zmacs command 263
  - List Methods (m-X) Zmacs command 378
  - tv: **list-mouse-buttons-mixin** flavor 364
  - Argument Lists 269
  - Lists in compiled code files 139
  - zl: **load** function 301
  - zl: **load-and-save-patches** function 212
  - compiler: **load-compiler-warnings** function 124
  - Load Compiler Warnings (m-X) Zmacs command 123, 309
  - Load File (m-X) Zmacs command 301
  - Compiling and Loading a File 301
  - Loading and Compiling Systems 175
  - Loading patches 197, 210
  - Loading System Definitions That Use Logical Pathnames 180
  - Loading System Definitions That Use Physical Pathnames 185
  - load-patches** function 211
  - Load Patches Command 211
  - load-system** function 176
  - load-system** Keywords 176
  - Load System Command 175
  - Load System command 180
  - Load-time dependency 161
  - dbg: **loc** function 64
  - Show Local Command 34
  - Show Monitored Locations Command 57
  - Electric Shift Lock Mode (m-X) Zmacs command 236
  - Creating a logical host 181
  - Sys:site; *Logical-host.Translations* File 182



Loading System Definitions That Use  
 Fill  
 Debugger read-eval-print  
 Lisp read-eval-print  
 Debugger breakpoints vs. break  
 Break  
 Entering a Break  
 Set  
 Logical Pathnames 180  
 Long Comment (m-X) Zmacs command 249  
 Long Documentation (m-sh-D) Zmacs  
 command 265, 267  
 Long-form Module Specifications 158  
 loop 9, 13  
 loop 13  
 loops 57  
 loops vs. Debugger breakpoints 13  
 Loop With SUSPEND, c-SUSPEND 13  
 Lowercase (m-X) Zmacs command 234

**M****M****M**

Meters ( M) 102  
 Query Replace ( m-%) Zmacs command 284  
 Edit Definition ( m-.) Zmacs command 266, 377, 378  
 Indent For Comment (c-; or m-;) Zmacs command 249  
 m-< 41  
 m-> 38  
 m-B 45  
 m-C 55  
 c-X  
 Evaluate Minibuffer ( m-ESCAPE) Zmacs command 303  
 m-I Debugger command 63  
 m-L 34  
 Indent New Comment Line ( m-LINE) Zmacs command 249  
 m-N 39  
 Down Comment Line ( m-N) Zmacs command 249  
 m-P 40  
 Up Comment Line ( m-P) Zmacs command 249  
 Compile Changed Definitions Of Buffer ( m-sh-C) Zmacs command 299  
 Long Documentation ( m-sh-D) Zmacs command 265, 267  
 Evaluate Changed Definitions Of Buffer ( m-sh-E) Zmacs command 303  
 Macro Expand Expression All ( m-sh-M) Zmacs command 332  
 Push Pop Point Explicit ( m-SPACE) Zmacs command 286  
 Entering the Debugger With m-SUSPEND, c-m-SUSPEND 13  
 Save Region ( m-W) Zmacs command 288  
 Abort Patch ( m-X) 209  
 Add Patch ( m-X) 206  
 Add Patch Changed Definitions ( m-X) 207  
 Add Patch Changed Definitions of Buffer ( m-X) 206  
 Finish Patch ( m-X) 208  
 Recompile Patch ( m-X) 209  
 Reload Patch ( m-X) 209  
 Resume Patch ( m-X) 209  
 Select Patch ( m-X) 207  
 Show Patches ( m-X) 207  
 Start Patch ( m-X) 205  
 Start Private Patch ( m-X) 205  
 Atom Word Mode ( m-X) Zmacs command 236  
 Auto Fill Mode ( m-X) Zmacs command 236  
 Compile Buffer ( m-X) Zmacs command 299  
 Compile Changed Definitions ( m-X) Zmacs command 299  
 Compile File ( m-X) Zmacs command 301  
 Compile Region ( m-X) Zmacs command 117  
 Compiler Warnings ( m-X) Zmacs command 123, 309  
 Deinstall Macro ( m-X) Zmacs command 292  
 Describe Flavor ( m-X) Zmacs command 377  
 Dired ( m-X) Zmacs command 271  
 Disassemble ( m-X) Zmacs command 335  
 Edit Callers ( m-X) Zmacs command 270

Edit Changed Definitions ( m-X) Zmacs command 282  
 Edit Changed Definitions Of Buffer ( m-X) Zmacs command 282  
 Edit Combined Methods ( m-X) Zmacs command 378  
 Edit Compiler Warnings ( m-X) Zmacs command 123, 309  
 Edit File Warnings ( m-X) Zmacs command 123  
 Edit Methods ( m-X) Zmacs command 293, 378  
 Electric Shift Lock Mode ( m-X) Zmacs command 236  
 Evaluate And Replace Into Buffer ( m-X) Zmacs command 303  
 Evaluate Buffer ( m-X) Zmacs command 303  
 Evaluate Changed Definitions ( m-X) Zmacs command 303  
 Evaluate Into Buffer ( m-X) Zmacs command 303  
 Fill Long Comment ( m-X) Zmacs command 249  
 Find Unbalanced Parentheses ( m-X) Zmacs command 252  
 Function Apropos ( m-X) Zmacs command 266  
 Insert Buffer ( m-X) Zmacs command 291  
 Insert File ( m-X) Zmacs command 291  
 Install Macro ( m-X) Zmacs command 292  
 Install Mouse Macro ( m-X) Zmacs command 292  
 Lisp Mode ( m-X) Zmacs command 236  
 List Callers ( m-X) Zmacs command 263, 270  
 List Changed Definitions ( m-X) Zmacs command 282  
 List Changed Definitions Of Buffer ( m-X) Zmacs command 282  
 List Combined Methods ( m-X) Zmacs command 378  
 List Matching Lines ( m-X) Zmacs command 284  
 List Matching Symbols ( m-X) Zmacs command 263  
 List Methods ( m-X) Zmacs command 378  
 Load Compiler Warnings ( m-X) Zmacs command 123, 309  
 Load File ( m-X) Zmacs command 301  
 Multiple Edit Callers ( m-X) Zmacs command 270  
 Multiple List Callers ( m-X) Zmacs command 270  
 Name Last Kbd Macro ( m-X) Zmacs command 292  
 Print Modifications ( m-X) Zmacs command 282  
 Reparse Attribute List ( m-X) Zmacs command 234  
 Select All Buffers As Tag Table ( m-X) Zmacs command 284  
 Select System As Tag Table ( m-X) Zmacs command 284  
 Set Backspace ( m-X) Zmacs command 234  
 Set Base ( m-X) Zmacs command 234  
 Set Fonts ( m-X) Zmacs command 234  
 Set Key ( m-X) Zmacs command 292  
 Set Lowercase ( m-X) Zmacs command 234  
 Set Nofill ( m-X) Zmacs command 234  
 Set Package ( m-X) Zmacs command 234  
 Set Patch File ( m-X) Zmacs command 234  
 Set Tab Width ( m-X) Zmacs command 234  
 Set Vsp ( m-X) Zmacs command 234  
 Source Compare ( m-X) Zmacs command 282  
 Source Compare Merge ( m-X) Zmacs command 282  
 Split Screen ( m-X) Zmacs command 293  
 Tags Query Replace ( m-X) Zmacs command 284  
 Tags Search ( m-X) Zmacs command 284  
 Trace ( m-X) Zmacs command 323, 325  
 Update Attribute List ( m-X) Zmacs command 234  
 View Directory ( m-X) Zmacs command 271  
 Where Is Symbol ( m-X) Zmacs command 263  
 m-Y input editor command 306  
 Yank Pop ( m-Y) Zmacs command 288  
 Files That  
   macro 111  
 #m sharp-sign reader macro 137  
 #q sharp-sign reader macro 137  
 compile-flavor-methods macro 369

**defflavor** macro 345, 349  
**Inhibit-style-warnings** macro 128  
**meter:with-monitoring** macro 223  
**sct:define-system-operation** macro 172  
**zl:if-for-lisp** macro 137  
**zl:if-for-maclisp** macro 137  
**zl:if-for-maclisp-else-lisp** macro 137  
**zl:if-in-lisp** macro 137  
**zl:if-in-maclisp** macro 138  
Start Kbd Macro (c-X ) Zmacs command 292  
End Kbd Macro (c-X ) Zmacs command 292  
Call Last Kbd Macro (c-X E) Zmacs command 292  
Macro Expand Expression All (m-sh-M) Zmacs command 332  
Macro Expand Expression (c-sh-M) Zmacs command 332  
Deinstall Macro (m-X) Zmacs command 292  
Install Macro (m-X) Zmacs command 292  
Install Mouse Macro (m-X) Zmacs command 292  
Name Last Kbd Macro (m-X) Zmacs command 292  
Expanding Macros 332  
Keyboard Macros 292  
Send mail about patch 208  
Mail Bug Report Command 62  
Mailing a bug report 61  
Maintaining Large Programs 141  
**:maintaining-sites** Option For **defsystem** 152  
System maintenance 197  
Major and Minor Modes 236  
Major version 201  
Major version number 197  
**tv: make-blinker** function 369  
**si: make-hardcopy-stream** function 359  
**fs: make-logical-pathname-host** 181  
**compiler: make-message-obsolete** special form 129  
**compiler: make-obsolete** special form 129  
**meter: make-pc-array** function 221  
**:batch** option for **make-system** 309  
**make-system** function 309  
**tv: make-window** function 344, 354, 359, 364  
Making Patches 203  
**meter: map-over-functions-in-bucket** function 223  
Set Pop Mark (c-SPACE) Zmacs command 286  
Swap Point And Mark (c-X c-X) Zmacs command 286  
Mark Definition (c-m-H) Zmacs command 288  
Mark Whole (c-X H) Zmacs command 291  
List Matching Lines (m-X) Zmacs command 284  
List Matching Symbols (m-X) Zmacs command 263  
Mentioned defaults 29  
[Inspect] in System menu 93  
[ARGPDL] **trace** menu item 323  
[Attributes] System menu item 377  
[Break after] **trace** menu item 323  
[Break before] **trace** menu item 323  
[Clear] Inspector menu item 95  
[Cond after] **trace** menu item 323  
[Cond before] **trace** menu item 323  
[Cond break after] **trace** menu item 323, 329  
[Cond break before] **trace** menu item 323, 329  
[Conditional] **trace** menu item 323  
[DeCache] Inspector menu item 95

- [Edit Screen] System menu item 294, 364
- [Edit] System menu item 233
- [Edit] Window Debugger menu item 310
- [Error] trace menu item 323, 329
- [Exit] Inspector menu item 95, 335
- [Inspect] System menu item 335
- [Modify] Inspector menu item 95, 335
- [Per Process] trace menu item 323
- [Print after] trace menu item 323
- [Print before] trace menu item 323
- [Print] trace menu item 323
- [Retry] Window Debugger menu item 310
- [Return] Inspector menu item 95, 335
- [Set \] Inspector menu item 95
- [Split Screen] System menu item 294, 364
- [Step] trace menu item 323, 325
- [Trace] System menu item 323, 325
- [Untrace] trace menu item 323
- [Wherein] trace menu item 323
- The Inspector Menu Pane 95
- Source Compare Merge (M-X) Zmacs command 282
- :fasd-form message 139
- Peek Help Message 99
- meter:expand-range function 222
- meter:function-name-with-escapes function 223
- meter:function-range function 223
- meter:list-functions-in-bucket function 223
- meter:make-pc-array function 221
- meter:map-over-functions-in-bucket function 223
- meter:monitor-all-functions function 222
- meter:monitor-between-functions function 222
- meter:print-functions-in-bucket function 223
- meter:range-of-bucket function 223
- meter:report function 222
- meter:setup-monitor function 222
- meter:start-monitor function 222
- meter:stop-monitor function 222
- meter:with-monitoring macro 223
- PC Metering 221
- Program Counter Metering 219
- Meters (M) 102
- Inspecting a select method 97
- :proceed method 369
- :report method 369
- :who-line-documentation-string method 364
- :case method combination 369
- :daemon method combination 345, 349, 354
- :send-command method of lgp::basic-lgp-stream 357
- :send-coordinates method of lgp::basic-lgp-stream 357
- :operation-handled-p method of sl:vanilla-flavor 378
- :which-operations method of sl:vanilla-flavor 378
- :any-tyl method of tv:any-tyl-mixin 364
- :draw-line method of tv:graphics-mixin 239, 349
- :change-of-size-or-margins method of tv:sheet 349
- :init method of tv:sheet 349
- :inside-size method of tv:sheet 349
- :refresh method of tv:sheet 349, 364
- Methods 378
- Daemon methods 349
- Primary methods 345, 349, 354
- Deriving Methods for Tools and Techniques 227

- Edit Methods (m-X) Zmacs command 293, 378
  - Edit Combined Methods (m-X) Zmacs command 378
  - List Methods (m-X) Zmacs command 378
  - List Combined Methods (m-X) Zmacs command 378
- mexp** function 332
- Minibuffer 232
  - c-? Zmacs minibuffer command 232
  - COMPLETE Zmacs minibuffer command 232
  - END Zmacs minibuffer command 232
  - HELP Zmacs minibuffer command 232
  - RETURN Zmacs minibuffer command 232
  - SPACE Zmacs minibuffer command 232
  - Evaluate Minibuffer (m-ESCAPE) Zmacs command 303
    - :**minimum-height** init option for **tv:essential-window** 349
    - :**minimum-width** init option for **tv:essential-window** 349
- Major and Minor Modes 236
- Minor version 201
- Minor version number 197, 203
- Miscellaneous Debugger Commands 63
- Miscellaneous Debugging Aids 91
- Missing Package Prefix 21
- Supplying a Mixin to Position the Figure 345
  - A Mode (m-X) Zmacs command 236
  - Atom Word Mode (m-X) Zmacs command 236
  - Auto Fill Mode (m-X) Zmacs command 236
  - Electric Shift Lock Mode (m-X) Zmacs command 236
  - Lisp Mode (m-X) Zmacs command 236
- Major and Minor Modes 236
- Peek Modes 101
- Show System Modifications Command 214
- Print Modifications (m-X) Zmacs command 282
- Modified Two Windows (c-X 4) Zmacs command 293
- Program Development:
  - Modifying the Output Module 344
  - [Modify] Inspector menu item 95, 335
  - Modularity 344
  - Module 145
  - Anonymous module 153, 156, 159
  - :**compile-satisfies-load** Option For **:module** 164
  - :**distribute-binaries** Option For **:module** 166
  - :**distribute-sources** Option For **:module** 166
  - :**in-order-to** Option For **:module** 161
  - :**package** Option For **:module** 159
  - Program Development: Modifying the Output Module 344
  - Root module 162
  - :**root-module** Option For **:module** 162
  - :**source-category** Option For **:module** 166
  - :**type** Option For **:module** 159
  - :**uses-definitions-from** Option For **:module** 161
  - :**module** Keyword Options 159
  - :**module** option for **defsystem** 158
  - Module Dependencies 154
  - Calculation Module for the Sample Program 383
  - Output Module for the Sample Program 403
  - Module group 156
  - :**module-group** clause 156
  - defsystem** Modules 153
  - Dependencies among modules 154
  - Module specification 159
  - Long-form Module Specifications 158

- Short-form Module Specifications 156
- User-defined Module Types 171
- Table of Module Types and Operations 169
- meter:** **monitor-all-functions** function 222
- meter:** **monitor-between-functions** function 222
- Show Monitored Locations Command 57
- Monitor facility functions 63
- dbg:** **monitor-instance-variable** function 65
- dbg:** **monitor-location** function 64
- Monitor traps 52
- Monitor Variable Command 54
- Motion 37
- Mouse 19
- Mouse 364
- Mouse Capabilities 10
- Mouse clicks 364
- Mouse cursor as an arrow 95
- Mouse documentation string 364
- Mouse in the Debugger 27
- mouse in the Inspector 95, 96
- Mouse Macro (m-X) Zmacs command 292
- Mouse-sensitive Debugger output 10, 19, 27
- Move To Previous Point (c-m-SPACE) Zmacs command 286
- Moving around in the stack 37
- Moving Text 286
- Moving Through Text 286
- Multiple Buffers 293
- Multiple Edit Callers (m-X) Zmacs command 270
- Multiple List Callers (m-X) Zmacs command 270
- zl:** **multiple-value** special form 349
- Using Multiple Windows 293
- Files That Maclisp Must Compile 137
- Debugger Commands for Stack
- Entering a Debugger Command with the Arrow Window: Interaction, Processes, and the Overview of Debugger
- Using the Install

## N

## N

## N

- Network (N) 103
- Inspecting a named structure 97
- Name Last Kbd Macro (m-X) Zmacs command 292
- Names 266
- Format of patch file names 201
- Names of Patch Files 201
- Network (N) 103
- Indent New Comment Line (m-LINE) Zmacs command 249
- Creating a New File 234
- Indent New Line (LINE) Zmacs command 251
- Next Frame Command 39
- :nil option for trace 323
- :arg :value :both** nil trace Options 76
- Set Nofill (m-X) Zmacs command 234
- note-private-patch** function 210
- Major version number 197
- Minor version number 197, 203
- Numbers in compiled code files 139
- Numeric arguments in accelerators 18

Compiled function object 109  
 Inspecting a compiled code object 97  
 Finding Out About Objects 260  
   Inspecting objects 96  
   Compiled code objects in compiled code files 139  
**compiler:** **obsolete-function-warning-switch** variable 133  
   Obtaining Information About a System 213  
   Obtaining Information on System Versions 215  
   Advising One Function Within Another 83  
   One Window (c-X 1) Zmacs command 293  
**compiler:** **open-code-map-switch** variable 133  
   Open Get Register (c-X G) Zmacs command 291  
   **operation-handled-p** method of  
     **si:vanilla-flavor** 378  
   Complex operations 172  
**defsystem** Operations 168  
   Generic operations 344  
 Table of Module Types and Operations 169  
   Other Operations on Systems 187  
   User-defined Operations on Systems 172  
   Optimizer feature of the compiler 135  
 Compiler Source-Level Optimizers 135  
   **:argpdl trace** Option 75  
   **:break trace** Option 74  
   c-sh-P proceed option 21  
   **:cond trace** Option 75  
   **:entrycond trace** Option 75  
   **:entryprint trace** Option 75  
   **:entry trace** Option 76  
   **:error trace** Option 74  
   **:exitbreak trace** Option 74  
   **:exitcond trace** Option 75  
   **:exitprint trace** Option 76  
   **:exit trace** Option 76  
   **:per-process trace** Option 75  
   **:print trace** Option 76  
   s-sh-C proceed option 21  
   **:step trace** Option 75  
   **:wherein trace** Option 75  
   **:default-init-plist** option for **defflavor** 349  
   **:documentation** option for **defflavor** 377  
**:gettable-instance-variables** option for **defflavor** 345  
**:initable-instance-variables** option for **defflavor** 357, 359  
   **:required-flavors** option for **defflavor** 349  
   **:required-methods** option for **defflavor** 345  
   **:advertised-in** Option For **defsystem** 151  
   **:bug-reports** Option For **defsystem** 151  
**:default-module-type** Option For **defsystem** 148  
   **:default-package** Option For **defsystem** 147  
   **:default-pathname** Option For **defsystem** 148  
   **:distribute-binaries** Option For **defsystem** 152  
   **:distribute-sources** Option For **defsystem** 152  
   **:initializations** Option For **defsystem** 150  
   **:initial-status** Option For **defsystem** 151  
   **:journal-directory** Option For **defsystem** 149  
   **:maintaining-sites** Option For **defsystem** 152  
   **:module** option for **defsystem** 158  
**:package-override** Option For **defsystem** 147  
   **:parameters** Option For **defsystem** 150  
   **:patchable** Option For **defsystem** 149, 197, 200

*patch-atom* argument to **:patchable** option for **defsystem** 201  
**:pathname-default** option for **defsystem** 197, 200  
**:pretty-name** Option For **defsystem** 146  
**:short-name** Option For **defsystem** 146  
**:source-category** Option For **defsystem** 152  
**:batch** option for **make-system** 309  
**:compile-satisfies-load** Option For **:module** 164  
**:distribute-binaries** Option For **:module** 166  
**:distribute-sources** Option For **:module** 166  
**:in-order-to** Option For **:module** 161  
**:package** Option For **:module** 159  
**:root-module** Option For **:module** 162  
**:source-category** Option For **:module** 166  
**:type** Option For **:module** 159  
**:uses-definitions-from** Option For **:module** 161  
**:arg** option for **trace** 76  
**:arg** option for **trace** 323  
**:argpdl** option for **trace** 323  
**:both** option for **trace** 76  
**:both** option for **trace** 323  
**:break** option for **trace** 323, 329  
**:cond** option for **trace** 323  
**:entry** option for **trace** 323  
**:entrycond** option for **trace** 323  
**:entryprint** option for **trace** 323  
**:error** option for **trace** 323, 329  
**:exit** option for **trace** 323  
**:exit** option for **trace** 76  
**:exitbreak** option for **trace** 323, 329  
**:exitcond** option for **trace** 323  
**:exitprint** option for **trace** 323  
**:nil** option for **trace** 323  
**:per-process** option for **trace** 323  
**:print** option for **trace** 323  
**:step** option for **trace** 323, 325  
**:value** option for **trace** 76  
**:value** option for **trace** 323  
**:wherein** option for **trace** 323  
**:function** option for **tv:choose-variable-values** 369  
**:edges-from** **init** option for **tv:essential-window** 349  
**:expose-p** **init** option for **tv:essential-window** 349  
**:minimum-height** **init** option for **tv:essential-window** 349  
**:minimum-width** **init** option for **tv:essential-window** 349  
**:process** **init** option for **tv:process-mixin** 364  
**:blinker-p** **init** option for **tv:sheet** 349  
**:arg :value :both nil trace** Options 76  
**Debugger Proceed and Restart** Options 12  
**defsystem** Options 146  
**:module** Keyword Options 159  
**Using Debugger Proceed and Restart** Options 20  
**Show Proceed** Options Command 48  
**Options To trace** 74  
**:argpdl** option to **trace** 75  
**:break** option to **trace** 74  
**:cond** option to **trace** 75  
**:entry** option to **trace** 76  
**:entrycond** option to **trace** 75  
**:entryprint** option to **trace** 75  
**:error** option to **trace** 74  
**:exitbreak** option to **trace** 74  
**:exitcond** option to **trace** 75



- :exitprint** option to **trace** 76
- :print** option to **trace** 76
- :step** option to **trace** 75
- :wherein** option to **trace** 75
- Organization of Patch Files 200
- Organization of Tools and Techniques 228
- Other Displays 295
- Other Operations on Systems 187
- Scroll
  - Other Window (c-m-V) Zmacs command 293
  - Other Window (c-X 0) Zmacs command 293
- Displaying Zmacs and
  - Other Windows 294
  - Finding Out About Existing Code 260
  - Finding Out About Functions 265
  - Finding Out About Objects 260
  - Finding Out About Pathnames 271
  - Finding Out About Symbols 263
  - Finding Out About Variables 265
- Commenting
  - Out Code 314
- Program Development: Design and Figure
  - Outline 238
  - Outlining the Figure 240
- Controlling the Format Of **trace**
  - Output 77
  - Flavors for LGP
    - Output 357
  - Mouse-sensitive Debugger
    - output 10, 19, 27
  - Simple Screen
    - Output 239
  - trace**
    - output 77
- Program Development: Modifying the
  - Output Module 344
  - Graphic
    - Output Module for the Sample Program 403
    - Output of the Sample Program 425
    - Overriding Variable-defined-but-never-referenced Warnings 131
    - Overview of Debugger Commands 7
    - Overview of Debugger Evaluation Environment 9
    - Overview of Debugger Help Facilities 10
    - Overview of Debugger Mouse Capabilities 10
    - Overview of Peek 99
    - Overview of the Debugger 5
- Other Window (c-X 0) Zmacs command 293

## P

## P

## P

- Processes ( P) 101
  - :package** Option For **:module** 159
  - Set
    - Package (m-X) Zmacs command 234
    - :package-override** Option For **defsystem** 147
  - Supplying a Missing
    - Package Prefix 21
    - Packages 234, 240, 263, 270
  - The Inspector History
    - Pane 95
  - The Inspector Inspection
    - Pane 96
  - The Inspector Interaction
    - Pane 95
  - The Inspector Menu
    - Pane 95
  - Inspection
    - Pane Display 97
    - :parallel** clause 153, 156
    - :parameters** Option For **defsystem** 150
  - Balancing
    - Parentheses 252
  - Find Unbalanced
    - Parentheses (m-X) Zmacs command 252
    - Patch 197
    - Current
      - patch 207
    - In-progress
      - patch 207
    - Send mail about
      - patch 208
  - :patchable** Option For **defsystem** 149, 197, 200
  - :patchable** option for **defsystem** 201
  - patch-atom* argument to



- Describe Variable At Point (c-sh-V) Zmacs command 265
- Push Pop Point Explicit (m-SPACE) Zmacs command 286
- Yank Pop (m-Y) Zmacs command 288
- Set Pop Mark (c-SPACE) Zmacs command 286
- Push Pop Point Explicit (m-SPACE) Zmacs command 286
- Jump To Saved Position (c-X J) Zmacs command 286
- Save Position (c-X S) Zmacs command 286
- A Mixin to Position the Figure 345
- Supplying a Missing Package Prefix 21
- Preparing to Write Code 233
- Prerequisites to Tools and Techniques 227
- :pretty-name** Option For **defsystem** 146
- Select Previous Buffer (c-m-L) Zmacs command 286
- Previous Frame Command 40
- Move To Previous Point (c-m-SPACE) Zmacs command 286
- Primary methods 345, 349, 354
- :print** option for **trace** 323
- :print** option to **trace** 76
- [Print after] **trace** menu item 323
- [Print before] **trace** menu item 323
- print-compiler-warnings** function 124
- Print compiler warnings database 123
- meter:** **print-functions-in-bucket** function 223
- Print Modifications (m-X) Zmacs command 282
- zl:** **print-system-modifications** function 216
- [Print] **trace** menu item 323
- :print trace** Option 76
- Start Private Patch (m-X) 205
- :proceed** method 369
- Debugger Proceed and Restart Options 12
- Using Debugger Proceed and Restart Options 20
- Proceed Command 51
- Proceed handlers 12, 20
- Proceeding 369
- Proceeding and Restarting in the Debugger 20
- c-sh-P proceed option 21
- s-sh-C proceed option 21
- Show Proceed Options Command 48
- Proceed Trap on Call Command 55
- Proceed types 310, 369
- :process** init option for **tv:process-mixin** 364
- Processes 364
- processes 99
- Processes, and the Mouse 364
- Processes (P) 101
- :process** init option for **tv: process-mixin** 364
- tv: process-mixin** flavor 364
- [Per Process] **trace** menu item 323
- progn** 111
- Program 383
- Graphic Output of the Sample Program 425
- Output Module for the Sample Program 403
- The Peek Program 99
- Program Counter Metering 219
- Program Development: Design and Figure Outline 238
- Program Development: Drawing Stripes 252
- Program Development: Modifying the Output Module 344
- Program Development: Refining Stripe Density and Spacing 272

Caveat to Introduction to Program Development Tools and Techniques 225  
 Debugging Lisp Programs 309  
 Maintaining Large Programs 141  
 Debugger command Program Strategy 238  
 Evaluation environment command prompt 7, 11, 17  
**compiler:top-level-form** **prompt-and-read** function 369  
 prompts 22  
 property 116  
 Purpose of Tools and Techniques 227  
 Push Pop Point Explicit (m-SPACE) Zmacs command 286  
 Put Register (c-X X) Zmacs command 291  
 Putting Data in Compiled Code Files 139

## Q

## Q

## Q

**zl:** **query-io** variable 369  
 Query Replace (m-%) Zmacs command 284  
 Query Replace (m-X) Zmacs command 284  
 Quick Arglist (c-sh-A) Zmacs command 269  
 Quit 103  
 Quit (c-Z) Zmacs command 310

## R

## R

## R

**meter:** **range-of-bucket** function 223  
 REFRESH r Dired (c-X D) Zmacs command 271  
 #m sharp-sign reader macro 137  
 #q sharp-sign reader macro 137  
 Debugger read-eval-print loop 9, 13  
 Lisp read-eval-print loop 13  
 Editing, Hardcopying, Reap-Protecting, and Releasing Systems 187  
**sct:** **reap-protect-system** function 188  
 Rebound Variable Bindings During Evaluation 24  
 Special Characters Recognized by the Inspector 97  
 Using Recompile Patch (m-X) 209  
 Program Development: Recursive Debugger Invocations 25  
 Refining Stripe Density and Spacing 272  
 REFRESH key 19, 34  
**:refresh** method of **tv:sheet** 349, 364  
 REFRESHr Dired (c-X D) Zmacs command 271  
 region 95  
 Line Region (c-m-\) Zmacs command 251  
 Indent Region (c-sh-C) Zmacs command 299  
 Compile Region (c-sh-E) Zmacs command 303  
 Evaluate Region (m-W) Zmacs command 288  
 Save Region (m-X) Zmacs command 117  
 Compile region to patch file 203  
 Open Get Register (c-X G) Zmacs command 291  
 Put Register (c-X X) Zmacs command 291  
 Using Registers 291  
 Reinvoke Command 51  
**sct:** **release-system** function 189  
 Editing, Hardcopying, Reap-Protecting, and Releasing Systems 187  
 Reload Patch (m-X) 209  
 Reparse Attribute List (m-X) Zmacs command 234  
 Replace (c-%) Zmacs command 284

Evaluate And Query Tags Query Searching and Mailing a bug  
**meter:**  
 Mail Bug  
 Show  
 Proceeding and Debugger Proceed and Using Debugger Proceed and  
 Using ABORT And  
 Debugger functions to

Replace Into Buffer (m-X) Zmacs command 303  
 Replace (m-%) Zmacs command 284  
 Replace (m-X) Zmacs command 284  
 Replacing 284  
 report 61  
**report** function 222  
**:report** method 369  
 Report Command 62  
**:required-flavors** option for **defflavor** 349  
**:required-methods** option for **defflavor** 345  
 Resources 359  
 Rest Argument Command 35  
 Restart handlers 12, 20, 310  
 Restarting in the Debugger 20  
 Restart Options 12  
 Restart Options 20  
 Restart Trap on Call Command 56  
 RESUME 303, 329  
 RESUME in the Debugger 21  
 RESUME key 12, 20, 51  
 RESUME key and traps 52  
 Resume Patch (m-X) 209  
 [Retry] Window Debugger menu item 310  
 RETURN key 40  
 RETURN Zmacs minibuffer command 232  
 Return Command 51  
 [Return] Inspector menu item 95, 335  
 return values in current stack frame 63  
 Reverse Search (c-R) Zmacs command 284  
 Root module 162  
**:root-module** Option For **:module** 162  
**rubout-handler** 24

## S

## S

## S

Servers ( S) 102  
 Calculation Module for the Graphic Output of the Output Module for the  
 Jump To  
 Converting LGP to Split Simple [Edit] [Split]  
 s-sh-C proceed option 21  
 Sample Program 383  
 Sample Program 425  
 Sample Program 403  
 Save Compiler Warnings Command 123  
 Saved Position (c-X J) Zmacs command 286  
 Save Position (c-X S) Zmacs command 286  
 Save Region (m-W) Zmacs command 288  
 Scope of Tools and Techniques 227  
 Screen Coordinates 354  
 Screen (m-X) Zmacs command 293  
 Screen Output 239  
 Screen] System menu item 294, 364  
 Screen] System menu item 294, 364  
 Scroll Other Window (c-m-V) Zmacs command 293  
 SCT 143  
**sct:check-system-patch-file-version** function 217  
**sct:define-module-type** function 171  
**sct:define-system-operation** macro 172  
**sct:designate-system-version** function 189  
**sct:edit-system** function 187  
**sct:get-all-system-input-files** function 217  
**sct:get-release-version** function 216  
**sct:get-system-input-and-output-defsystem-files** function 216

- sct:get-system-input-and-output-source-files** function 216
- sct:get-system-version** function 215
- sct:hardcopy-system** function 187
- sct:patch-loaded-p** function 216
- sct:patch-system-pathname** function 202
- sct:reap-protect-system** function 188
- sct:release-system** function 189
- sct:set-system-source-file** 181, 183, 185
- sct:set-system-source-file** function 181
- sct:set-system-status** function 188
- sct:system-version-info** function 215
- Reverse Search (c-R) Zmacs command 284
- Incremental Search (c-S) Zmacs command 284
- Searching and Replacing 284
- Tags Search (m-X) Zmacs command 284
- SELECT E 233
- SELECT I 93, 335
- Select Activity command 233
- Select All Buffers As Tag Table (m-X) Zmacs command 284
- Select Buffer (c-X B) Zmacs command 286
- Inspecting a select method 97
- Select Patch (m-X) 207
- Select Previous Buffer (c-m-L) Zmacs command 286
- Select System As Tag Table (m-X) Zmacs command 284
- :send-command** method of
  - lgp::basic-lgp-stream** 357
- :send-coordinates** method of
  - lgp::basic-lgp-stream** 357
- Send mail about patch 208
- zwei: \*send-mail-about-patch\*** 208
- :serial** clause 153, 156
- Servers (S) 102
- Set Backspace (m-X) Zmacs command 234
- Set Base (m-X) Zmacs command 234
- Set Breakpoint Command 60
- Set Comment Column (c-X ;) Zmacs command 249
- Set Current Frame Command 41
- Set Fill Column (c-X F) Zmacs command 236
- Set Fonts (m-X) Zmacs command 234
- Set Key (m-X) Zmacs command 292
- fs: set-logical-pathname-host** 182
- Set Lowercase (m-X) Zmacs command 234
- Set Nofill (m-X) Zmacs command 234
- Set Package (m-X) Zmacs command 234
- Set Patch File (m-X) Zmacs command 234
- Set Pop Mark (c-SPACE) Zmacs command 286
- Set sleep time between updates Peek command 99
- Set Stack Size Command 63
- sct: set-system-source-file** 181, 183, 185
- sl: set-system-source-file** 181
- sct: set-system-source-file** function 181
- sct: set-system-status** function 188
- Set Tab Width (m-X) Zmacs command 234
- Setting Debugger breakpoints 57
- Set Trap on Call Command 56
- Set Trap on Exit Command 56

**meter:** **setup-monitor** function 222  
 Set Vsp (m-X) Zmacs command 234  
 [Set \] Inspector menu item 95  
 Kill Sexp (c-m-K) Zmacs command 288  
 Indent Sexp (c-m-Q) Zmacs command 251  
 Backward Kill Sexp (c-m-RUBOUT) Zmacs command 288  
 #m sharp-sign reader macro 137  
 #q sharp-sign reader macro 137  
**:blinker-p** init option for tv: **sheet** 349  
**:change-of-size-or-margins** method of tv: **sheet** 349  
     **:init** method of tv: **sheet** 349  
**:inside-size** method of tv: **sheet** 349  
**:refresh** method of tv: **sheet** 349, 364  
 tv: **sheet** flavor 349, 369  
 tv: **sheet-following-blinker** function 369  
 Electric Shift Lock Mode (m-X) Zmacs command 236  
 Short-form Module Specifications 156  
**:short-name** Option For **defsystem** 146  
 Show Arglist Command 32  
 Show Argument Command 32  
**dbg:** **\*show-backtrace\*** variable 66  
 Show Backtrace Command 45  
 Show Bindings Command 46  
 Show Breakpoints Command 61  
 Show Catch Blocks Command 46  
 Show Compiled Code Command 33  
 Show Compiler Warnings Command 123  
 Show Condition Handlers Command 47  
**:Show Frame** (c-L) 19  
 Show Frame Command 34  
 Show Function Command 34  
 Show Instruction Command 47  
 Show Lexical Environment Command 48  
 Show Local Command 34  
 Show Monitored Locations Command 57  
 Show Patches (m-X) 207  
 Show Proceed Options Command 48  
 Show Rest Argument Command 35  
 Show Source Code Command 35  
 Show Special Command 48  
 Show Stack Command 36  
 Show Standard Value Warnings Command 49  
 Show System Definition Command 213  
 Show System Modifications Command 214  
 Show System Plan Command 171, 215  
 Show Value Command 36  
**si:advise-1** function 81  
**si:advised-functions** variable 82  
**si:flavor-allowed-init-keywords** function 381  
**si:make-hardcopy-stream** function 359  
**si:set-system-source-file** 181  
**si:\*trace-bar-p\*** variable 78  
**si:\*trace-bar-rate\*** variable 78  
**si:\*trace-columns-per-level\*** variable 78  
**si:\*trace-old-style\*** variable 78  
**si:unadvise-1** function 81  
**si:unbin-file** function 98  
**si:vanilla-flavor** 378  
**:operation-handled-p** method of **si:vanilla-flavor** 378  
**:which-operations** method of **si:vanilla-flavor** 378  
**si:vanilla-flavor** flavor 378  
**signal** function 344, 369

- Defining Flavors to
  - Signal Conditions 369
- Tools for Compiling
  - Simple Screen Output 239
  - Single Functions 120
  - Single Step Command 61
  - Single Stepping 57
- Debugger Commands for Breakpoints and
  - si:pkg-user-package** 24
  - Size Command 63
  - Set Stack
    - sleep time between updates Peek command 99
  - Set File Types of Lisp
    - Source and Compiled Code Files 119
    - :source-category** Option For **defsystem** 152
    - :source-category** Option For **:module** 166
  - Show
    - Source Code Command 35
    - Source Compare (m-X) Zmacs command 282
    - Source Compare Merge (m-X) Zmacs command 282
    - source files 197
  - System Compiler
    - Source-Level Optimizers 135
    - SPACE Stepper command 325
    - SPACE Zmacs minibuffer command 232
- Program Development: Refining Stripe Density and Spacing 272
  - special** special form 114
  - Special Characters Recognized by the Inspector 97
  - Special Command 48
  - special commands 310
  - special form 79
  - special form 83
  - special form 135
  - special form 129
  - special form 129
  - special form 369
  - special form 167
  - special form 146, 197, 284
  - special form 240, 298
  - special form 364
  - special form 115
  - special form 114
  - special form 73, 323, 325, 329
  - special form 81
  - special form 84
  - special form 78, 323
  - special form 369
  - special form 359
  - special form 329
  - special form 298
  - special form 130
  - special form 131
  - special form 130
  - special form 349
  - special form 115
  - special form 359
  - Special variables 133
  - specification 159
  - Specifications 158
  - Specifications 156
  - Specifying compiler environments 137
  - Split Screen (m-X) Zmacs command 293
  - [Split Screen] System menu item 294, 364
  - stack 66
  - stack 37
  - Stack Command 38
  - Stack Command 36
  - Stack Command 41
- Backtrace of the call
  - Moving around in the
    - Bottom of
      - Show
    - Top of
- show
  - Debugger
    - advise**
    - advise-within**
    - compiler:add-optimizer**
    - compiler:make-message-obsolete**
    - compiler:make-obsolete**
    - condition-bind**
    - defsubsystem**
    - defsystem**
    - defvar**
    - error-restart-loop**
    - eval-when**
    - special**
    - trace**
    - unadvise**
    - unadvise-within**
    - untrace**
    - unwind-protect**
    - with-open-stream**
    - zl:break**
    - zl:defconst**
    - zl:\*expr**
    - zl:\*fexpr**
    - zl:\*lexpr**
    - zl:multiple-value**
    - zl:unspecial**
    - zl-user:defwindow-resource**
  - Module
    - Long-form Module
    - Short-form Module



- Current stack frame 31
- Debugger Commands for Viewing a Stack Frame 31
- Debugger functions to return values in current stack frame 63
- Inspecting a stack frame 97
- Debugger Commands for Stack Motion 37
- Set Stack Size Command 63
- zl: **standard-output** variable 303
- Show Standard Value Warnings Command 49
- Start Kbd Macro (c-X ()) Zmacs command 292
- meter: **start-monitor** function 222
- Start Patch (m-X) 205
- Start Private Patch (m-X) 205
- state 207
- Initial patch state 207
- In-progress patch status 99
- System status of active processes 99
- Display status of areas 99
- Display status of file system display 99
- Display status of hostat 99
- Display status of window area 99
- zl: **step** function 85, 325
- :step** option for **trace** 323, 325
- :step** option to **trace** 75
- Single Step Command 61
- Step facility 5
- step-form** variable 85
- Stepper 325
- c-B Stepper command 325
- c-E Stepper command 325
- c-N Stepper command 325
- c-U Stepper command 325
- c-X Stepper command 325
- HELP Stepper command 325
- SPACE Stepper command 325
- Stepping 303, 325
- Stepping 57
- Stepping 323
- Stepping Through an Evaluation 85
- Stepping through compiled code 57, 61
- [Step] **trace** menu item 323, 325
- :step trace** Option 75
- step-value** variable 85
- step-values** variable 85
- meter: **stop-monitor** function 222
- Supplying a Value to Store Permanently 21
- Program Strategy 238
- Stream 66
- Stream compiler 109
- Stream Compiler Handles Top-level Forms 111
- string 364
- strings 265, 267
- Stripe Density and Spacing 272
- Stripes 252
- structure 97
- Structure of the Compiler 109
- compiler: **style-checker** function 127
- Compiler Style Warnings 127
- Journal subdirectory 191, 197
- Summary of Compiler Actions on Code in a Zmacs Buffer 299
- Summary of Debugger Commands 67

- SUPER key bindings 12, 20
  - Supplying a Missing Package Prefix 21
  - Supplying a Value to Store Permanently 21
  - SUSPEND 303
- Entering a Break Loop With
  - SUSPEND, c-SUSPEND 13
  - Swap Point And Mark (c-X c-X) Zmacs command 286
- Compiler Switches 133
- Inspecting a
  - symbol 97
- Where Is
  - Symbol (m-X) Zmacs command 263
- Finding Out About
  - Symbols 263
  - Symbols in compiled code files 139
  - Symbols (m-X) Zmacs command 263
- List Matching
  - Symeval in Last Instance Command 49
- Keeping Track of Lisp
  - Syntax 247
  - Syntax attribute 235
  - sys:abort** flavor 364
  - sys:dump-forms-to-file** function 139
  - sys:file-local-declarations** variable 130
  - sys:trace-compile-flag** variable 77
  - Sys:site;Logical-host.Translations File 182
  - Sys:site;system-name.system 181
  - Sys:site;System-name.System File 181, 183
  - System 143
  - System 145
  - System 191
  - System 213
  - System As Tag Table (m-X) Zmacs command 284
  - System command 180
  - System Command 177
  - System command 180
  - System Command 175
  - System Construction Tool 143
  - System Construction Tool 143
  - System Declaration File 183, 191
  - System Definition Command 213
  - System Definitions That Use Logical Pathnames 180
  - System Definitions That Use Physical Pathnames 185
  - System directory 149
  - System-directory 149
  - system display 99
  - System (F) 102
  - System facility 143
  - System file 181
  - system information 99
  - System maintenance 197
  - System menu 93
  - System menu item 377
  - System menu item 233
  - System menu item 294, 364
  - System menu item 335
  - System menu item 294, 364
  - System menu item 323, 325
  - System Modifications Command 214
  - system-name.system 181
  - System-name.System File 181, 183
  - System Plan 171
  - System Plan Command 171, 215
  - systems 159
  - Systems 187
- Defining a
  - System 145
- Directories Associated with a
  - System 191
- Obtaining Information About a
  - System 213
  - System As Tag Table (m-X) Zmacs command 284
  - System command 180
  - System Command 177
  - System command 180
  - System Command 175
- Select
  - System As Tag Table (m-X) Zmacs command 284
- Compile
  - System command 180
- Compile
  - System Command 177
- Load
  - System command 180
- Load
  - System Command 175
- Introduction to the
  - System Construction Tool 143
  - System Construction Tool 143
  - System Declaration File 183, 191
  - System Definition Command 213
  - System Definitions That Use Logical Pathnames 180
  - System Definitions That Use Physical Pathnames 185
  - System directory 149
  - System-directory 149
  - system display 99
  - System (F) 102
  - System facility 143
  - System file 181
  - system information 99
  - System maintenance 197
  - System menu 93
  - System menu item 377
  - System menu item 233
  - System menu item 294, 364
  - System menu item 335
  - System menu item 294, 364
  - System menu item 323, 325
  - System Modifications Command 214
  - system-name.system 181
  - System-name.System File 181, 183
  - System Plan 171
  - System Plan Command 171, 215
  - systems 159
  - Systems 187
- Show
  - System Definition Command 213
- Loading
  - System Definitions That Use Logical Pathnames 180
- Loading
  - System Definitions That Use Physical Pathnames 185
  - System directory 149
  - System-directory 149
  - system display 99
  - System (F) 102
  - System facility 143
  - System file 181
  - system information 99
  - System maintenance 197
  - System menu 93
  - System menu item 377
  - System menu item 233
  - System menu item 294, 364
  - System menu item 335
  - System menu item 294, 364
  - System menu item 323, 325
  - System Modifications Command 214
  - system-name.system 181
  - System-name.System File 181, 183
  - System Plan 171
  - System Plan Command 171, 215
  - systems 159
  - Systems 187
- Display status of file
  - system display 99
  - System (F) 102
  - System facility 143
  - System file 181
  - system information 99
  - System maintenance 197
  - System menu 93
  - System menu item 377
  - System menu item 233
  - System menu item 294, 364
  - System menu item 335
  - System menu item 294, 364
  - System menu item 323, 325
  - System Modifications Command 214
  - system-name.system 181
  - System-name.System File 181, 183
  - System Plan 171
  - System Plan Command 171, 215
  - systems 159
  - Systems 187
- File
  - System (F) 102
  - System facility 143
  - System file 181
  - system information 99
  - System maintenance 197
  - System menu 93
  - System menu item 377
  - System menu item 233
  - System menu item 294, 364
  - System menu item 335
  - System menu item 294, 364
  - System menu item 323, 325
  - System Modifications Command 214
  - system-name.system 181
  - System-name.System File 181, 183
  - System Plan 171
  - System Plan Command 171, 215
  - systems 159
  - Systems 187
- Display
  - system information 99
  - System maintenance 197
  - System menu 93
  - System menu item 377
  - System menu item 233
  - System menu item 294, 364
  - System menu item 335
  - System menu item 294, 364
  - System menu item 323, 325
  - System Modifications Command 214
  - system-name.system 181
  - System-name.System File 181, 183
  - System Plan 171
  - System Plan Command 171, 215
  - systems 159
  - Systems 187
- [Inspect] in
  - System menu 93
- [Attributes]
  - System menu item 377
- [Edit]
  - System menu item 233
- [Edit Screen]
  - System menu item 294, 364
- [Inspect]
  - System menu item 335
- [Split Screen]
  - System menu item 294, 364
- [Trace]
  - System menu item 323, 325
- Show
  - System Modifications Command 214
- Sys:site;
  - system-name.system 181
- Sys:site;
  - System-name.System File 181, 183
- Show
  - System Plan 171
- Component
  - System Plan Command 171, 215
- Editing, Hardcopying, Reap-Protecting, and Releasing
  - systems 159
  - Systems 187

Loading and Compiling Systems 175  
   Other Operations on Systems 187  
     Updating systems 197  
 User-defined Operations on Systems 172  
   Distribute Systems Command 189  
     System source files 197  
     System status 99  
 Debugger Commands for System Transfer 61  
   File types of the system version-directory file 201  
     **sct:** **system-version-info** function 215  
     System versions 197  
     System Versions 215  
     S) Zmacs command 286

## T

## T

## T

c-X T 54  
   Indent For Lisp ( TAB or c-m-TAB) Zmacs command 251  
     Command table 17, 29  
     Global command table 7  
   Select All Buffers As Tag Table (m-X) Zmacs command 284  
   Select System As Tag Table (m-X) Zmacs command 284  
     Table of Module Types and Operations 169  
     tables 284  
   Tag Tab Width (m-X) Zmacs command 234  
   Set Tags Query Replace (m-X) Zmacs command 284  
     Tags Search (m-X) Zmacs command 284  
     Tag Table (m-X) Zmacs command 284  
     Tag Table (m-X) Zmacs command 284  
     Tag tables 284  
   Select All Buffers As Techniques 225  
   Select System As Techniques 227  
 Caveat to Program Development Tools and Techniques 228  
   Deriving Methods for Tools and Techniques 227  
   Features Described in Tools and Techniques 228  
   Introduction to Tools and Techniques 227  
   Organization of Tools and Techniques 228  
   Prerequisites to Tools and Techniques 227  
   Program Development Tools and Techniques 225  
   Purpose of Tools and Techniques 227  
   Scope of Tools and Techniques 227  
     **zl:** **terminal-io** variable 239, 344  
     Moving Text 286  
     Moving Through Text 286  
       Files That Maclisp Must Compile 137  
   Loading System Definitions That Use Logical Pathnames 180  
   Loading System Definitions That Use Physical Pathnames 185  
   Compiler Tools and Their Differences 117  
     **throw** function 52  
     Throw Command 52  
     Set sleep time between updates Peek command 99  
 Introduction to the System Construction Tool 143  
   System Construction Tool 143  
   Debugging tools 5  
   Caveat to Program Development Tools and Techniques 225  
   Deriving Methods for Tools and Techniques 227  
   Features Described in Tools and Techniques 228  
   Introduction to Tools and Techniques 227  
   Organization of Tools and Techniques 228  
   Prerequisites to Tools and Techniques 227  
   Program Development Tools and Techniques 225  
   Purpose of Tools and Techniques 227  
   Scope of Tools and Techniques 227

- Compiler
  - Tools and Their Differences 117
  - Tools for Compiling Code From the Editor Into Your World 117
  - Tools for Compiling Files 118
  - Tools for Compiling Single Functions 120
  - Tools: Using the Debugger 310
- compiler:**
  - top-level-form** property 116
- Controlling the Evaluation of
  - How the Stream Compiler Handles
    - The
      - Move
        - To Previous Point (c-m-SPACE) Zmacs command 286
        - To Saved Position (c-X J) Zmacs command 286
      - Jump
        - To trace 74
      - Options
        - :arg option for trace 76
        - :arg option for trace 323
        - :argpdl option for trace 323
        - :argpdl option to trace 75
        - :both option for trace 76
        - :both option for trace 323
        - :break option for trace 323, 329
        - :break option to trace 74
        - :cond option for trace 323
        - :cond option to trace 75
        - :entry option for trace 323
        - :entry option to trace 76
        - :entrycond option for trace 323
        - :entrycond option to trace 75
        - :entryprint option for trace 323
        - :entryprint option to trace 75
        - :error option for trace 323, 329
        - :error option to trace 74
        - :exit option for trace 323
        - :exit option for trace 76
        - :exitbreak option for trace 323, 329
        - :exitbreak option to trace 74
        - :exitcond option for trace 323
        - :exitcond option to trace 75
        - :exitprint option for trace 323
        - :exitprint option to trace 76
        - :nil option for trace 323
        - Options To trace 74
        - :per-process option for trace 323
        - :print option for trace 323
        - :print option to trace 76
        - :step option for trace 323, 325
        - :step option to trace 75
        - :value option for trace 76
        - :value option for trace 323
        - :wherein option for trace 323
        - :wherein option to trace 75
        - [ARGPDL] trace menu item 323
        - [Break after] trace menu item 323
        - [Break before] trace menu item 323
        - [Cond after] trace menu item 323
        - [Cond before] trace menu item 323
        - [Cond break after] trace menu item 323, 329
        - [Cond break before] trace menu item 323, 329
        - [Conditional] trace menu item 323
        - [Error] trace menu item 323, 329

|                                 |                                             |
|---------------------------------|---------------------------------------------|
| [Per Process]                   | trace menu item 323                         |
| [Print]                         | trace menu item 323                         |
| [Print after]                   | trace menu item 323                         |
| [Print before]                  | trace menu item 323                         |
| [Step]                          | trace menu item 323, 325                    |
| [Untrace]                       | trace menu item 323                         |
| [Wherein]                       | trace menu item 323                         |
| :argpd                          | trace Option 75                             |
| :break                          | trace Option 74                             |
| :cond                           | trace Option 75                             |
| :entry                          | trace Option 76                             |
| :entrycond                      | trace Option 75                             |
| :entryprint                     | trace Option 75                             |
| :error                          | trace Option 74                             |
| :exit                           | trace Option 76                             |
| :exitbreak                      | trace Option 74                             |
| :exitcond                       | trace Option 75                             |
| :exitprint                      | trace Option 76                             |
| :per-process                    | trace Option 75                             |
| :print                          | trace Option 76                             |
| :step                           | trace Option 75                             |
| :wherein                        | trace Option 75                             |
| :arg :value :both nil           | trace Options 76                            |
|                                 | trace output 77                             |
| Controlling the Format Of       | trace Output 77                             |
|                                 | trace special form 73, 323, 325, 329        |
| si:                             | *trace-bar-p* variable 78                   |
| si:                             | *trace-bar-rate* variable 78                |
| si:                             | *trace-columns-per-level* variable 78       |
| sys:                            | trace-compile-flag variable 77              |
|                                 | Trace facility 5                            |
|                                 | Trace (m-X) Zmacs command 323, 325          |
| si:                             | *trace-old-style* variable 78               |
|                                 | [Trace] System menu item 323, 325           |
|                                 | Tracing 323                                 |
|                                 | Tracing and Stepping 323                    |
| Disable Condition               | Tracing Command 54                          |
| Enable Condition                | Tracing Command 54                          |
|                                 | Tracing Function Execution 73               |
| Keeping                         | Track of Lisp Syntax 247                    |
| Debugger Commands for System    | Transfer 61                                 |
|                                 | Translations file 182                       |
| Debugger                        | trap 11                                     |
| Debugger                        | Trap Commands 52                            |
| Clear                           | Trap on Call Command 53                     |
| Proceed                         | Trap on Call Command 55                     |
| Restart                         | Trap on Call Command 56                     |
| Set                             | Trap on Call Command 56                     |
| Clear                           | Trap on Exit Command 53                     |
| Set                             | Trap on Exit Command 56                     |
|                                 | Traps 52                                    |
| ABORT key and                   | traps 52                                    |
| Monitor                         | traps 52                                    |
| RESUME key and                  | traps 52                                    |
| :any-tyi method of              | tv:any-tyi-mixin 364                        |
|                                 | tv:any-tyi-mixin flavor 364                 |
| :function option for            | tv:choose-variable-values 369               |
|                                 | tv:choose-variable-values function 359, 369 |
| :edges-from init option for     | tv:essential-window 349                     |
| :expose-p init option for       | tv:essential-window 349                     |
| :minimum-height init option for | tv:essential-window 349                     |



## V

## V

## V

**dbg:** **val** function 64  
**:value** option for **trace** 323  
**:value** option for **trace** 76  
**:arg** **:value :both nil trace** Options 76  
**Show** **Value** Command 36  
**values** variable 74  
**values** variable 329  
**values** in current stack frame 63  
**values** of instance variables 63  
**Value** to Store Permanently 21  
**Value** Warnings Command 49  
**:operation-handled-p** method of **si:** **vanilla-flavor** 378  
**:which-operations** method of **si:** **vanilla-flavor** 378  
**si:** **vanilla-flavor** flavor 378  
**applyhook** variable 89  
**arglist** variable 329  
**arglist** variable 74  
**compiler:compiler-verbose** variable 133  
**compiler:functions-defined** variable 130  
**compiler:functions-referenced** variable 130  
**compiler:inhibit-style-warnings-switch** variable 133  
**compiler:obsolete-function-warning-switch** variable 133  
**compiler:open-code-map-switch** variable 133  
**dbg:\*show-backtrace\*** variable 66  
**dbg:\*debug-io-override\*** variable 66  
**dbg:\*defer-package-dwim\*** variable 66  
**dbg:\*frame\*** variable 66  
**dbg:\*show-backtrace\*** variable 66  
**evalhook** variable 87  
**ignore** variable 130  
**si:advised-functions** variable 82  
**si:\*trace-bar-p\*** variable 78  
**si:\*trace-bar-rate\*** variable 78  
**si:\*trace-columns-per-level\*** variable 78  
**si:\*trace-old-style\*** variable 78  
**step-form** variable 85  
**step-value** variable 85  
**step-values** variable 85  
**sys:file-local-declarations** variable 130  
**sys:trace-compile-flag** variable 77  
**values** variable 329  
**values** variable 74  
**zl:all-special-switch** variable 133  
**zl:query-io** variable 369  
**zl:standard-output** variable 303  
**zl:terminal-io** variable 239, 344  
**Describe** **Variable At Point (c-sh-V)** Zmacs command 265  
**Rebound** **Variable Bindings During Evaluation** 24  
**Monitor** **Variable Command** 54  
**Unmonitor** **Variable Command** 57  
**Overriding** **Variable-defined-but-never-referenced Warnings** 131  
**Compiler** **variables** 130  
**Debugger** **Variables** 66  
**Examining values of instance** **variables** 63  
**Finding Out About** **Variables** 265  
**Instance** **variables** 345, 357, 359  
**Special** **variables** 133  
**Major** **version** 201  
**Minor** **version** 201  
**File types of the system** **version-directory file** 201

- Major version number 197
- Minor version number 197, 203
- Obtaining Information on System Versions 215
- System versions 197
- View Directory (m-X) Zmacs command 271
- Viewing a Stack Frame 31
- View Two Windows (c-X 3) Zmacs command 293
- vs. break loops 57
- vs. CP commands 7, 17, 29
- vs. Debugger breakpoints 13
- Vsp (m-X) Zmacs command 234

## W

## W

## W

- Windows ( W) 102
- Compiler warnings 293, 299
- Compiler Style Warnings 127
- Controlling Compiler Warnings 127
- Function-referenced-but-never-defined Warnings 130
- Overriding Variable-defined-but-never-referenced Warnings 131
- Save Compiler Warnings Command 123
- Show Compiler Warnings Command 123
- Show Standard Value Warnings Command 49
- Compiler Warnings Database 123
- Print compiler warnings database 123
- Update compiler warnings database 123
- Using the Compiler Warnings Database 309
- Compiler Warnings (m-X) Zmacs command 123, 309
- Edit Compiler Warnings (m-X) Zmacs command 123, 309
- Edit File Warnings (m-X) Zmacs command 123
- Load Compiler Warnings (m-X) Zmacs command 123, 309
- what-files-call** function 263
- :wherein** option for **trace** 323
- :wherein** option to **trace** 75
- [Wherein] **trace** menu item 323
- :wherein trace** Option 75
- where-is** function 263
- Where Is Symbol (m-X) Zmacs command 263
- :which-operations** method of **si:vanilla-flavor** 378
- who-calls** function 263
- Whole (c-X H) Zmacs command 291
- :who-line-documentation-string** method 364
- Whoppers 349
- Width (m-X) Zmacs command 234
- window 359, 364
- Window 349
- window** flavor 344, 349
- window area 99
- Window (c-m-V) Zmacs command 293
- Window (c-X 1) Zmacs command 293
- Window (c-X 0) Zmacs command 293
- Window Debugger 5, 310
- Window Debugger 61, 63
- :Window Debugger (c-m-W) command 19
- Window Debugger Command 63
- Window Debugger menu item 310
- Window Debugger menu item 310
- Window: Interaction, Processes, and the Mouse 364
- Windows 294
- Windows 377
- Windows 343



Using Multiple Windows 293  
 Two Windows (c-X 2) Zmacs command 293  
 View Two Windows (c-X 3) Zmacs command 293  
 Modified Two Windows (c-X 4) Zmacs command 293  
 Windows (w) 102  
 Advising One Function Within Another 83  
**meter:** **with-monitoring** macro 223  
**with-open-stream** special form 359  
 Atom Word Mode (m-X) Zmacs command 236  
 How the Inspector Works 93  
 Tools for Compiling Code From the Editor Into Your World 117  
 Preparing to Write Code 233  
 Writing and Editing Code 231

**X**

**X**  
 Put Register (c-X X) Zmacs command 291

**X****Y**

**Y**  
 Killing and Yank (c-Y) Zmacs command 288  
 Yanking 288  
 Yank Pop (m-Y) Zmacs command 288  
 Tools for Compiling Code From the Editor Into Your World 117

**Y****Y****Z**

**Z**  
**zl:all-special-switch** variable 133  
**zl:apropos** function 263  
**zl:break** special form 329  
**zl:dbg** function 14, 329  
**zl:defconst** special form 298  
**zl:\*expr** special form 130  
**zl:\*fexpr** special form 131  
**zl:if-for-lisp** macro 137  
**zl:if-for-maclisp** macro 137  
**zl:if-for-maclisp-else-lisp** macro 137  
**zl:if-in-lisp** macro 137  
**zl:if-in-maclisp** macro 138  
**zl:\*lexpr** special form 130  
**zl:l1starray** function 260  
**zl:load** function 301  
**zl:load-and-save-patches** function 212  
**zl:multiple-value** special form 349  
**zl:peek** function 99  
**zl:pkg-goto** function 240  
**zl:plist** function 263  
**zl:print-system-modifications** function 216  
**zl:query-io** variable 369  
**zl:standard-output** variable 303  
**zl:step** function 85, 325  
**zl:terminal-io** variable 239, 344  
**zl:typep** function 377  
**zl:unspecial** special form 115  
 Entering the Debugger With **break** And **zl:dbg** Functions 14  
**zl:ibase** 24  
**zl:read-preserve-delimiters** 24  
**zl-user:defwindow-resource** special form 359  
**zl-user:describe-flavor** function 377

**Z****Z**

|                                                 |                                     |                               |
|-------------------------------------------------|-------------------------------------|-------------------------------|
|                                                 | Using                               | Zmacros 231                   |
|                                                 | Using The HELP Key in               | Zmacros 231                   |
|                                                 | Displaying                          | Zmacros and Other Windows 294 |
|                                                 | Compiling Code in a                 | Zmacros Buffer 299            |
| Summary of Compiler Actions on Code in a        |                                     | Zmacros Buffer 299            |
| Atom Word Mode (m-X)                            |                                     | Zmacros command 236           |
| Auto Fill Mode (m-X)                            |                                     | Zmacros command 236           |
| Backward Kill Sexp (c-m-RUBOUT)                 |                                     | Zmacros command 288           |
| Beep (c-G)                                      |                                     | Zmacros command 286           |
| Brief Documentation (c-sh-D)                    |                                     | Zmacros command 265, 267      |
| c-sh-C                                          |                                     | Zmacros command 117           |
| Call Last Kbd Macro (c-X E)                     |                                     | Zmacros command 292           |
| Compile Buffer (m-X)                            |                                     | Zmacros command 299           |
| Compile Changed Definitions (m-X)               |                                     | Zmacros command 299           |
| Compile Changed Definitions Of Buffer (m-sh-C)  |                                     | Zmacros command 299           |
| Compile File (m-X)                              |                                     | Zmacros command 301           |
| Compile Region (c-sh-C)                         |                                     | Zmacros command 299           |
| Compile Region (m-X)                            |                                     | Zmacros command 117           |
| Compiler Warnings (m-X)                         |                                     | Zmacros command 123, 309      |
| Deinstall Macro (m-X)                           |                                     | Zmacros command 292           |
| Describe Flavor (m-X)                           |                                     | Zmacros command 377           |
| Describe Variable At Point (c-sh-V)             |                                     | Zmacros command 265           |
| Dired (m-X)                                     |                                     | Zmacros command 271           |
| Disassemble (m-X)                               |                                     | Zmacros command 335           |
| Display Directory (c-X c-D)                     |                                     | Zmacros command 271           |
| Down Comment Line (m-N)                         |                                     | Zmacros command 249           |
| Edit Callers (m-X)                              |                                     | Zmacros command 270           |
| Edit Changed Definitions (m-X)                  |                                     | Zmacros command 282           |
| Edit Changed Definitions Of Buffer (m-X)        |                                     | Zmacros command 282           |
| Edit Combined Methods (m-X)                     |                                     | Zmacros command 378           |
| Edit Compiler Warnings (m-X)                    |                                     | Zmacros command 123, 309      |
| Edit Definition (m-.)                           |                                     | Zmacros command 266, 377, 378 |
| Edit File Warnings (m-X)                        |                                     | Zmacros command 123           |
| Edit Methods (m-X)                              |                                     | Zmacros command 293, 378      |
| Electric Shift Lock Mode (m-X)                  |                                     | Zmacros command 236           |
| End Kbd Macro (c-X )                            |                                     | Zmacros command 292           |
| Evaluate And Replace Into Buffer (m-X)          |                                     | Zmacros command 303           |
| Evaluate Buffer (m-X)                           |                                     | Zmacros command 303           |
| Evaluate Changed Definitions (m-X)              |                                     | Zmacros command 303           |
| Evaluate Changed Definitions Of Buffer (m-sh-E) |                                     | Zmacros command 303           |
| Evaluate Into Buffer (m-X)                      |                                     | Zmacros command 303           |
| Evaluate Minibuffer (m-ESCAPE)                  |                                     | Zmacros command 303           |
| Evaluate Region (c-sh-E)                        |                                     | Zmacros command 303           |
| Fill Long Comment (m-X)                         |                                     | Zmacros command 249           |
| Find File (c-X c-F)                             |                                     | Zmacros command 234           |
| Find Unbalanced Parentheses (m-X)               |                                     | Zmacros command 252           |
| Function Apropos (m-X)                          |                                     | Zmacros command 266           |
| HELP                                            |                                     | Zmacros command 292           |
| Incremental Search (c-S)                        |                                     | Zmacros command 284           |
| Indent For Comment (c-; or m-;)                 |                                     | Zmacros command 249           |
| Indent For Lisp (TAB or c-m-TAB)                |                                     | Zmacros command 251           |
| Indent New Comment Line (m-LINE)                |                                     | Zmacros command 249           |
| Indent New Line (LINE)                          |                                     | Zmacros command 251           |
| Indent Region (c-m-\)                           |                                     | Zmacros command 251           |
| Indent Sexp (c-m-Q)                             |                                     | Zmacros command 251           |
| Insert Buffer (m-X)                             |                                     | Zmacros command 291           |
| Insert File (m-X)                               |                                     | Zmacros command 291           |
| Install Macro (m-X)                             |                                     | Zmacros command 292           |
| Install Mouse Macro (m-X)                       |                                     | Zmacros command 292           |
| Jump To Saved Position (c-X J)                  |                                     | Zmacros command 286           |
|                                                 | <b>zl-user:undefsystem</b> function | 166                           |

|                                          |               |          |
|------------------------------------------|---------------|----------|
| Kill Comment (c-m- ;)                    | Zmacs command | 249      |
| Kill Sexp (c-m-K)                        | Zmacs command | 288      |
| Lisp Mode (m-X)                          | Zmacs command | 236      |
| List Callers (m-X)                       | Zmacs command | 263, 270 |
| List Changed Definitions (m-X)           | Zmacs command | 282      |
| List Changed Definitions Of Buffer (m-X) | Zmacs command | 282      |
| List Combined Methods (m-X)              | Zmacs command | 378      |
| List Matching Lines (m-X)                | Zmacs command | 284      |
| List Matching Symbols (m-X)              | Zmacs command | 263      |
| List Methods (m-X)                       | Zmacs command | 378      |
| Load Compiler Warnings (m-X)             | Zmacs command | 123, 309 |
| Load File (m-X)                          | Zmacs command | 301      |
| Long Documentation (m-sh-D)              | Zmacs command | 265, 267 |
| Macro Expand Expression All (m-sh-M)     | Zmacs command | 332      |
| Macro Expand Expression (c-sh-M)         | Zmacs command | 332      |
| Mark Definition (c-m-H)                  | Zmacs command | 288      |
| Mark Whole (c-X H)                       | Zmacs command | 291      |
| Modified Two Windows (c-X 4)             | Zmacs command | 293      |
| Move To Previous Point (c-m-SPACE)       | Zmacs command | 286      |
| Multiple Edit Callers (m-X)              | Zmacs command | 270      |
| Multiple List Callers (m-X)              | Zmacs command | 270      |
| Name Last Kbd Macro (m-X)                | Zmacs command | 292      |
| One Window (c-X 1)                       | Zmacs command | 293      |
| Open Get Register (c-X G)                | Zmacs command | 291      |
| Other Window (c-X O)                     | Zmacs command | 293      |
| Print Modifications (m-X)                | Zmacs command | 282      |
| Push Pop Point Explicit (m-SPACE)        | Zmacs command | 286      |
| Put Register (c-X X)                     | Zmacs command | 291      |
| Query Replace (m-%)                      | Zmacs command | 284      |
| Quick Arglist (c-sh-A)                   | Zmacs command | 269      |
| Quit (c-Z)                               | Zmacs command | 310      |
| REFRESHr Dired (c-X D)                   | Zmacs command | 271      |
| Reparse Attribute List (m-X)             | Zmacs command | 234      |
| Replace (c-%)                            | Zmacs command | 284      |
| Reverse Search (c-R)                     | Zmacs command | 284      |
| Save Position (c-X S)                    | Zmacs command | 286      |
| Save Region (m-W)                        | Zmacs command | 288      |
| Scroll Other Window (c-m-V)              | Zmacs command | 293      |
| Select All Buffers As Tag Table (m-X)    | Zmacs command | 284      |
| Select Buffer (c-X B)                    | Zmacs command | 286      |
| Select Previous Buffer (c-m-L)           | Zmacs command | 286      |
| Select System As Tag Table (m-X)         | Zmacs command | 284      |
| Set Backspace (m-X)                      | Zmacs command | 234      |
| Set Base (m-X)                           | Zmacs command | 234      |
| Set Comment Column (c-X ;)               | Zmacs command | 249      |
| Set Fill Column (c-X F)                  | Zmacs command | 236      |
| Set Fonts (m-X)                          | Zmacs command | 234      |
| Set Key (m-X)                            | Zmacs command | 292      |
| Set Lowercase (m-X)                      | Zmacs command | 234      |
| Set Nofill (m-X)                         | Zmacs command | 234      |
| Set Package (m-X)                        | Zmacs command | 234      |
| Set Patch File (m-X)                     | Zmacs command | 234      |
| Set Pop Mark (c-SPACE)                   | Zmacs command | 286      |
| Set Tab Width (m-X)                      | Zmacs command | 234      |
| Set Vsp (m-X)                            | Zmacs command | 234      |
| Source Compare (m-X)                     | Zmacs command | 282      |
| Source Compare Merge (m-X)               | Zmacs command | 282      |
| Split Screen (m-X)                       | Zmacs command | 293      |
| Start Kbd Macro (c-X ()                  | Zmacs command | 292      |
| Swap Point And Mark (c-X c-X)            | Zmacs command | 286      |
| Tags Query Replace (m-X)                 | Zmacs command | 284      |

Tags Search (m-X) Zmacs command 284  
 Trace (m-X) Zmacs command 323, 325  
 Two Windows (c-X 2) Zmacs command 293  
 Up Comment Line (m-P) Zmacs command 249  
 Update Attribute List (m-X) Zmacs command 234  
 View Directory (m-X) Zmacs command 271  
 View Two Windows (c-X 3) Zmacs command 293  
 Where Is Symbol (m-X) Zmacs command 263  
 Yank (c-Y) Zmacs command 288  
 Yank Pop (m-Y) Zmacs command 288  
 Zmacs Command Completion 232  
 Fundamental Zmacs editing commands 23  
   c-? Zmacs minibuffer command 232  
 COMPLETE Zmacs minibuffer command 232  
   END Zmacs minibuffer command 232  
   HELP Zmacs minibuffer command 232  
   RETURN Zmacs minibuffer command 232  
   SPACE Zmacs minibuffer command 232  
 zwei:\*send-mail-about-patch\* 208

[Set \] Inspector menu item 95

