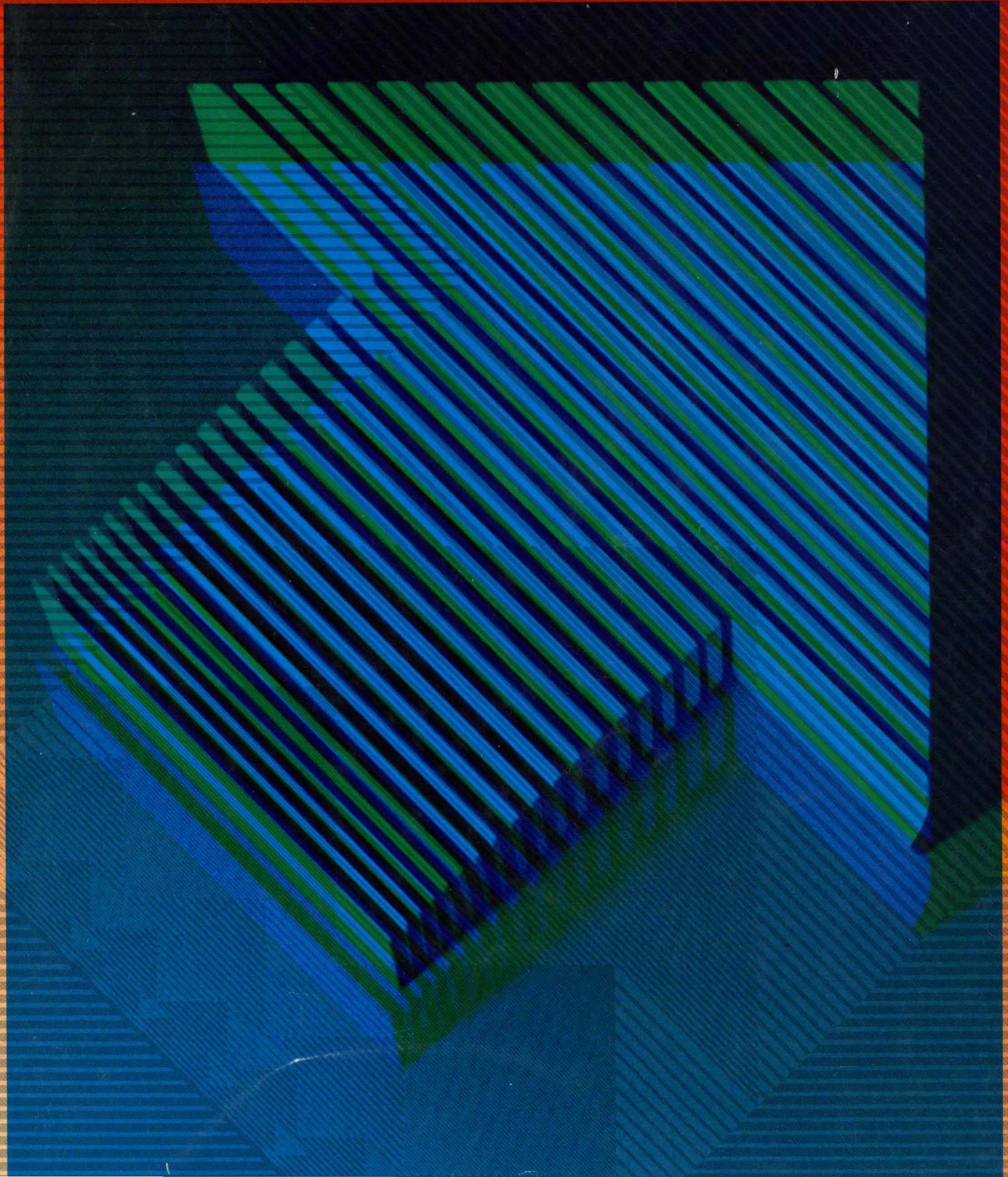
4 Program Development Utilities

symbolics



4 Program Development Utilities

symbolics

Cambridge, Massachusetts

Program Development Utilities
 # 996045

February 1985

This document corresponds to Release 6.0 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 © 1985, 1984, 1983, 1982, 1981, 1980 Symbolics, Inc. All Rights Reserved.
Font Library Copyright © 1984 Bitstream Inc. All Rights Reserved.

Symbolics, Symbolics 3600, Symbolics 3670, Symbolics 3640, SYMBOLICS-LISP, ZETALISP, MACSYMA, S-GEOMETRY, S-PAINT, and S-RENDER are trademarks of Symbolics, Inc.

Restricted Rights Legend
Use, duplication, or disclosure by the government is 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.

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

Text typography: Century Schoolbook and Helvetica produced on Symbolics 3600-family computers from Bitstream, Inc., outlines; text masters printed on Symbolics LGP-1 Laser Graphics Printers.
Cover design: Schafer/LaCasse
Cover printer: W.E. Andrews Co., Inc.
Text printer: ZBR Publications, Inc.

Printed in the USA.

Printing year and number: 87 86 85 9 8 7 6 5 4 3 2 1

Table of Contents

Page

I. Program Development Tools and Techniques **1**

1. Introduction **3**

1.1 Purpose 3
1.2 Prerequisites 3
1.3 Scope 3
1.4 Method 3
1.5 Features 4
1.6 Organization 4

2. Writing and Editing Code **7**

2.1 Before You Begin 7
2.1.1 HELP 7
2.1.2 Completion 8
2.2 Getting Started 9
2.2.1 Entering Zmacs 9
2.2.2 Creating a File 9
2.2.3 File Attribute Lists 10
2.2.4 Major and Minor Modes 12
2.3 Program Development: Design and Figure Outline 13
2.3.1 Program Strategy 13
2.3.2 Simple Screen Output 14
2.3.3 Outlining the Figure 16
2.4 Keeping Track of Lisp Syntax 23
2.4.1 Comments 23
2.4.2 Aligning Code 26
2.4.3 Balancing Parentheses 26
2.5 Program Development: Drawing Stripes 27
2.6 Finding Out About Existing Code 35
2.6.1 Objects 35
2.6.2 Symbols 38
2.6.3 Variables 39
2.6.4 Functions 40
2.6.5 Pathnames 45
2.7 Program Development: Refining Stripe Density and Spacing 45
2.8 Editing Code 56
2.8.1 Identifying Changed Code 56
2.8.2 Searching and Replacing 57
2.8.3 Moving Text 59

2.8.4 Keyboard Macros 64
2.8.5 Using Multiple Windows 65

3. Compiling and Evaluating Lisp **69**

3.1 Compiling Lisp Code 70
3.1.1 Compiling Code in a Zmacs Buffer 70
3.1.2 Compiling and Loading a File 73
3.2 Evaluating Lisp Code 75
3.2.1 Evaluation and the Editor 75
3.2.2 Lisp Input Editing 77

4. Debugging Lisp Programs **79**

4.1 The Compiler Warnings Database 79
4.2 The Debugger 80
4.3 Commenting Out Code 83
4.4 Tracing and Stepping 92
4.4.1 Tracing 92
4.4.2 Stepping 94
4.5 Breakpoints 98
4.6 Expanding Macros 100
4.7 The Inspector 104

5. Using Flavors and Windows **111**

5.1 Program Development: Modifying the Output Module 112
5.1.1 A Mixin to Position the Figure 113
5.1.2 The Basic Arrow Window 116
5.1.3 Converting Lgp to Screen Coordinates 121
5.1.4 Flavors for Lgp Output 124
5.1.5 The Top-level Function 126
5.1.6 The Arrow Window: Interaction, Processes, and the Mouse 129
5.1.7 Signalling Conditions 133
5.2 Programming Aids for Flavors and Windows 141
5.2.1 General Information on Flavors 141
5.2.2 Methods 142
5.2.3 Init Keywords 144

6. Calculation Module for the Sample Program **147**

7. Output Module for the Sample Program **165**

8. Graphic Output of the Sample Program **185**

II. Maintaining Large Programs **187**

9. Introduction to the System Facility 189

10. Defining a System 191

10.1 **defsystem** Modules 198
10.2 **defsystem** Transformations 201
10.2.1 Interaction Between **defsystem** Transformations and **make-system** 202
10.2.2 List of **defsystem** Transformations 209
10.2.3 **:skip defsystem** Macro 213
10.3 Adding New Options to **defsystem** 214

11. Loading the System Definition 217

11.1 Loading System Definitions That Use Logical Pathnames 217
11.1.1 Sys:site;*System-name*.System File 217
11.1.2 Sys:site;*Logical-host*.Translations File 218
11.1.3 System Declaration File 218
11.2 Loading System Definitions That Use Physical Pathnames 219

12. Making a System 221

12.1 Adding New Keywords to **make-system** 227

13. Patch Facility 231

13.1 Types of Patch Files 233
13.1.1 System Version-directory File 233
13.1.2 Patch Directory File 234
13.1.3 Individual Patch Files 235
13.1.4 Organization of Patch Files 235
13.1.5 Names of Patch Files 236
13.2 Making Patches 238
13.2.1 Start Patch (m-X) 239
13.2.2 Start Private Patch (m-X) 240
13.2.3 Add Patch (m-X) 240
13.2.4 Add Patch Changed Definitions of Buffer (m-X) 241
13.2.5 Add Patch Changed Definitions (m-X) 241
13.2.6 Select Patch (m-X) 242
13.2.7 View Patches (m-X) 242
13.2.8 Finish Patch (m-X) 242
13.2.9 Abort Patch (m-X) 243
13.2.10 Resume Patch (m-X) 243
13.2.11 Recompile Patch (m-X) 243
13.2.12 Reload Patch (m-X) 244

13.3 Loading Patches 244

14. Getting Information About a System **249**

15. Functions That Operate on a System **251**

15.1 Changing the Status of a Patchable System 251

III. Debugger **253**

16. Entering the Debugger **255**

16.1 Entering the Debugger by Causing an Error 255
16.1.1 Error Display 255
16.2 Entering the Debugger with m-SUSPEND 256
16.3 Entering the Debugger with the **dbg** Function 256

17. How to Use the Debugger **259**

17.1 Evaluating a Form in the Debugger 259
17.1.1 Rebound Variable Bindings During Evaluation 260
17.2 Exiting From the Debugger: Abort 261
17.3 Debugger Help 261
17.4 Proceeding From the Error in the Debugger: Resume 262
17.5 Examining the Current Stack Frame in the Debugger 262
17.6 Examining Stack Frames with Debugger Backtrace Commands 262
17.7 Debugger Commands for Stack Manipulation 263
17.8 Debugger Commands That Call Other Systems 264
17.8.1 Entering the Editor From the Debugger 264
17.8.2 Sending a Bug Report 264
17.8.3 Entering the Display Debugger 265
17.9 Debugger Commands for Information Display 265
17.10 Debugger Commands That Trap on Frame Exit 266
17.11 Debugger Commands for Dynamic Breakpoints and Stepping Through Compiled Code 267
17.12 Debugger Functions 267
17.13 Debugger Variables 268

18. Summary of Debugger Commands **271**

19. Summary of Debugging Aids **273**

20. Tracing Function Execution **275**
- 20.1 Options to **trace** 276
- 20.2 Controlling the Format of **trace** Output 279
- 20.3 Untracing Function Execution 280

21. Advising a Function **281**
- 21.1 Designing the Advice 283
- 21.2 **:around** Advice 284
- 21.3 Advising One Function Within Another 285

22. Stepping Through an Evaluation **287**

23. evalhook **289**
- 23.1 **applyhook** 290

IV. The Inspector **293**

24. Using the Inspector **295**
- 24.1 How the Inspector Works 295
- 24.2 Entering and Leaving the Inspector 295
- 24.3 The Inspector Interaction Pane 297
- 24.4 The Inspector History Pane 297
- 24.5 The Inspector Menu Pane 298
- 24.6 The Inspector Inspection Pane 298
  - 24.6.1 Inspection Pane Display 299
- 24.7 Special Characters Recognized by the Inspector 300
- 24.8 Examining a Compiled Code File 300

V. The Peek Program **301**

25. Peek **303**

VI. The Compiler **305**

26. Introduction to the Compiler **307**
- 26.1 How to Invoke the Compiler 307

27. Structure of the Compiler **309**
- 27.1 How the Stream Compiler Handles Top-level Forms 310
  - 27.1.1 Controlling the Evaluation of Top-level Forms 314
- 27.2 Function Compiler 315
- 27.3 **bin** File Dumper 316
- 27.4 Compiler Tools and Their Differences 316
  - 27.4.1 Tools for Compiling Code From the Editor Into Your World 316

27.4.2 Tools for Compiling Files 317
27.4.3 Tools for Compiling Single Functions 318

28. Compiler Warnings Database **321**

29. Controlling Compiler Warnings **323**

29.1 Compiler Style Warnings 323
29.2 Function-referenced-but-never-defined Warnings 324
29.2.1 Overriding Variable-defined-but-never-referenced Warnings 326

30. Compiler Source-level Optimizers **327**

31. Files That Maclisp Must Compile **329**

32. Putting Data in Compiled Code Files **331**

Index **333**

List of Figures

Figure 1. Program output with only the outlines of the arrows in the figure. 22
Figure 2. Program output with stripes of even spacing and density. 36
Figure 3. Program output with stripes of varying spacing and density. 55
Figure 4. Using multiple windows to test the program while viewing the source code. 67
Figure 5. Edit Compiler Warnings (m-X) splits the screen. The upper window contains compiler warnings. The lower window contains the source code. 81
Figure 6. The Display Debugger: inspecting the stack frame containing a call to **compute-dens**. 84
Figure 7. The Display Debugger: inspecting the variable ***x2***. 85
Figure 8. Output resulting from a faulty attempt to outline the small arrows recursively. 89
Figure 9. Output resulting from a faulty attempt to outline the small arrows recursively, with the second function call commented out. 90
Figure 10. Output resulting from a corrected attempt to outline the small arrows recursively, with the second function call commented out. 91
Figure 11. Output from the program with a bug in the function **draw-arrow-shaft-stripes**. 102
Figure 12. The Inspector window: inspecting an instance of a structure. 106
Figure 13. The Inspector window: inspecting an instance of a flavor. 108
Figure 14. The Inspector. 296

List of Tables

Table 1. Trace Menu Items and **trace** Options 95

PART I.

Program Development Tools and Techniques

1. Introduction

1.1 Purpose

In this document we introduce the Lisp programming environment of the Symbolics Lisp Machine. 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 on the Symbolics Lisp Machine. 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 Lisp Machine programming style of your own.

1.2 Prerequisites

This document is for you if you will be writing or maintaining Lisp programs and have recently begun to use a Symbolics Lisp Machine. The document will be most useful if you have some experience writing Lisp programs and are familiar with basic features of the Symbolics Lisp Machine. The document is not a survey of Symbolics Lisp Machine facilities, a reference manual, or a Lisp primer. You might find the following Symbolics publications especially helpful when reading this document:

- See the document *User's Guide to Symbolics Computers.*
- See the section "Program Development Help Facilities".

1.3 Scope

We focus in this document on interaction between programmers and the Symbolics Lisp Machine. We present some ways of using Symbolics Lisp Machine 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 document corresponds to the Symbolics 3600-family computers.

1.4 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.

1.5 Features

Symbolics developers have designed the Symbolics Lisp Machine to accommodate a relatively spontaneous and incremental programming style. Five Symbolics Lisp Machine 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.

1.6 Organization

The sequence of steps in developing a program on the Symbolics Lisp Machine is too complex to mirror in the linear organization of a document. We emphasize the cyclical course of program development, but we have organized the document 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 the Symbolics Lisp Machine. For some simple uses of flavors and windows and some programming aids for working with them: See the section "Using Flavors and Windows", page 111.

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 147. See the section "Output Module for the Sample Program", page 165.

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 185.

Many of the techniques and facilities we mention are helpful at more than one stage of program development. Conversely, the Symbolics Lisp Machine 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 on the Symbolics Lisp Machine.

In the sections of this document 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
```

2. Writing and Editing Code

Symbolics Lisp Machine programmers 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 Symbolics Lisp Machine 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 Lisp Machine 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 147. See the section "Output Module for the Sample Program", page 165. See the section "Graphic Output of the Sample Program", page 185.

2.1 Before You Begin

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 "Program Development Help Facilities".

2.1.1 HELP

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 contain a string you specify. (A refers to "apropos".)
HELP C	Displays the name and brief description of a command bound to a key you specify.
HELP D	Displays long 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. (W refers to "where".)
HELP SPACE	Repeats the last HELP command.

2.1.2 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, the word "Completion" appears in parentheses above the right side of the minibuffer.

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

SPACE	Completes words up to the current word.

HELP or c-?	Displays possible completions in the typeout area.
[*Zmacs Window* (R)]	Pops up a menu of possible completions.
COMPLETE	Displays the full name if possible.
RETURN or END	Confirms the name if possible, whether or not you have seen the full name.

2.2 Getting Started

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.

2.2.1 Entering Zmacs

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.

2.2.2 Creating a 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 "File Attribute Lists: Program Development Tools and Techniques", page 10. 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.

2.2.3 File Attribute Lists

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 **base** and **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 is assumed to be Zetalisp.
- If there is a Syntax: Common-Lisp attribute, 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 Zetalisp. 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 *attribute* (m-X)	Sets *attribute* for the current buffer. Queries whether or not to set *attribute* for the file and in

	the text attribute list. *attribute* is one of the following: 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.

2.2.4 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 23.

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**:

Host system	*File name*
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 m-F) operate on Lisp symbol names.
Auto Fill Mode (m-X)	Automatically breaks lines that extend beyond a preset fill column.
Set Fill Column (c-X F)	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.

2.3 Program Development: Design and Figure Outline

2.3.1 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 147. For the second file: See the section "Output Module for the Sample Program", page 165.

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 111. 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.

2.3.2 Simple Screen Output

For a discussion of the output routines: See the section "Using Flavors and Windows", page 111. 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 **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 **(:method tv:graphics-mixin :draw-line)** in *Programming the User Interface.*

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 **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:

```
(defflavor 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))))
```

2.3.3 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 147.

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 *Reference Guide to Symbolics-Lisp.*
- 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 *Reference Guide to Symbolics-Lisp.*
- 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 129.

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 70. 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 **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 1 shows the output of the program at this stage.

These six steps illustrate a pattern of incremental program development:

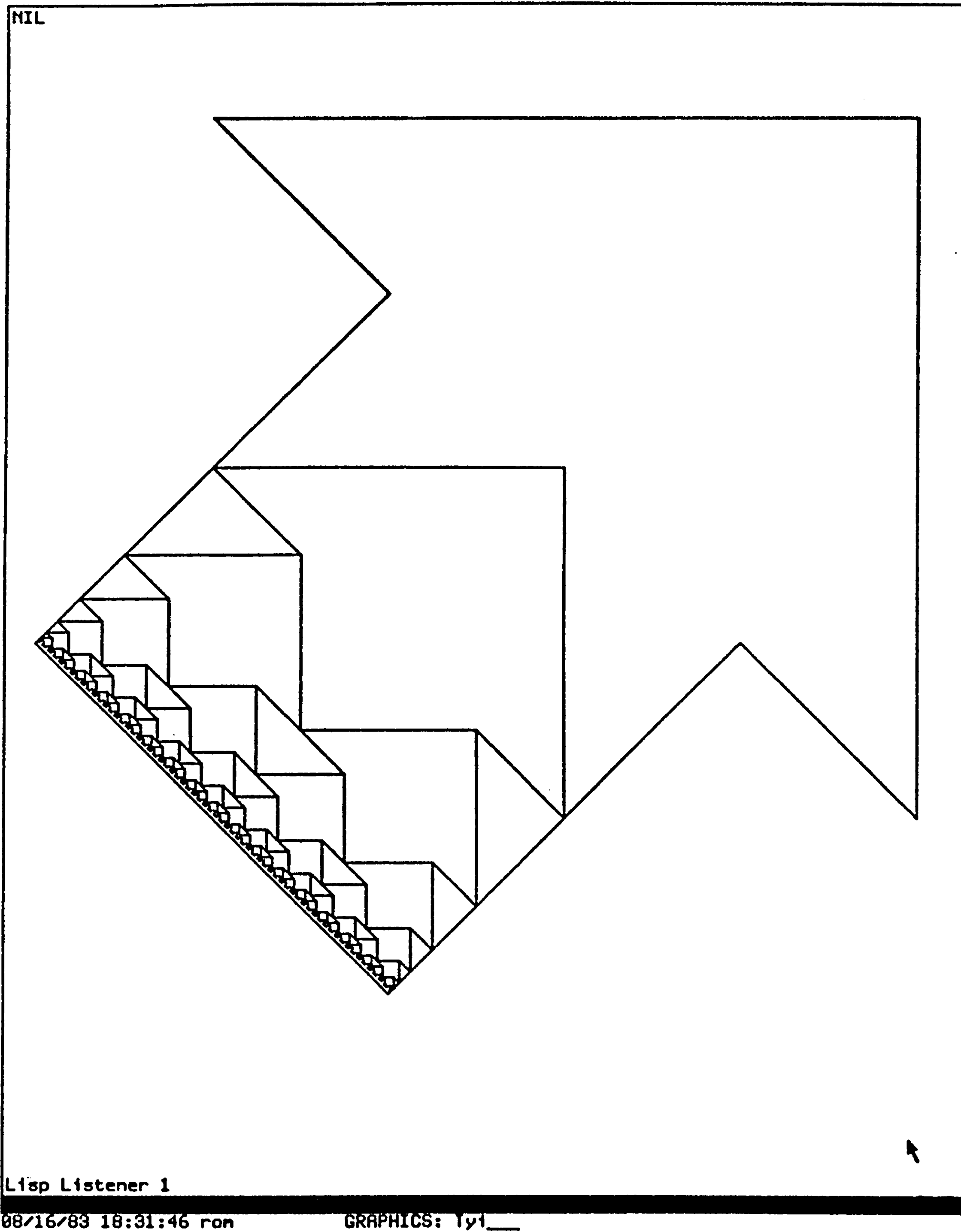


Figure 1. Program output with only the outlines of the arrows in the figure.

- 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 23. See the section "Editing Code: Program Development Tools and Techniques", page 56.
- 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 23. See the section "Editing Code: Program Development Tools and Techniques", page 56. See the section "Compiling and Evaluating Lisp", page 69.

2.4 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-m- modifiers. For example, Forward Sexp (c-m-F) moves forward to the end of a Lisp expression; End Of Definition (c-m-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 (m-F) places the cursor at the end of the name rather than before the first hyphen. See the section "Major and Minor Modes: Program Development Tools and Techniques", page 12.

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.*

2.4.1 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 35. 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 35. 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 69. See the section "Debugging Lisp Programs", page 79.

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-X) 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 (c-; or m-;)	Inserts or aligns a comment on the current line, beginning in the preset comment column.
Kill Comment (c-m-;)	Removes a comment from the current line.
Down Comment Line (m-N)	Moves to the comment column on the next line. Starts a comment if none is there.
Up Comment Line (m-P)	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 (m-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.

2.4.2 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

Indent New Line (LINE)	Adds a newline and indents as appropriate for the current level of Lisp structure.
Indent For Lisp (TAB or c-m-TAB)	Aligns the current line. If the line is blank, indents as appropriate for the current level of Lisp structure.
Indent Sexp (c-m-Q)	Aligns the Lisp expression following the cursor.
Indent Region (c-m-\)	Aligns the current region.

2.4.3 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.

2.5 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 2 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 35.
- 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: Program Development Tools and Techniques", page 56.

2.6 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 Lisp Machine 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.

2.6.1 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.

An interactive, window-oriented version of **describe** is the Inspector. See the section "The Inspector: Program Development Tools and Techniques", page 104.

describe does not display array elements. For that you can use the Inspector or **listarray**.

Example

```
(describe '*top-edge*)
```

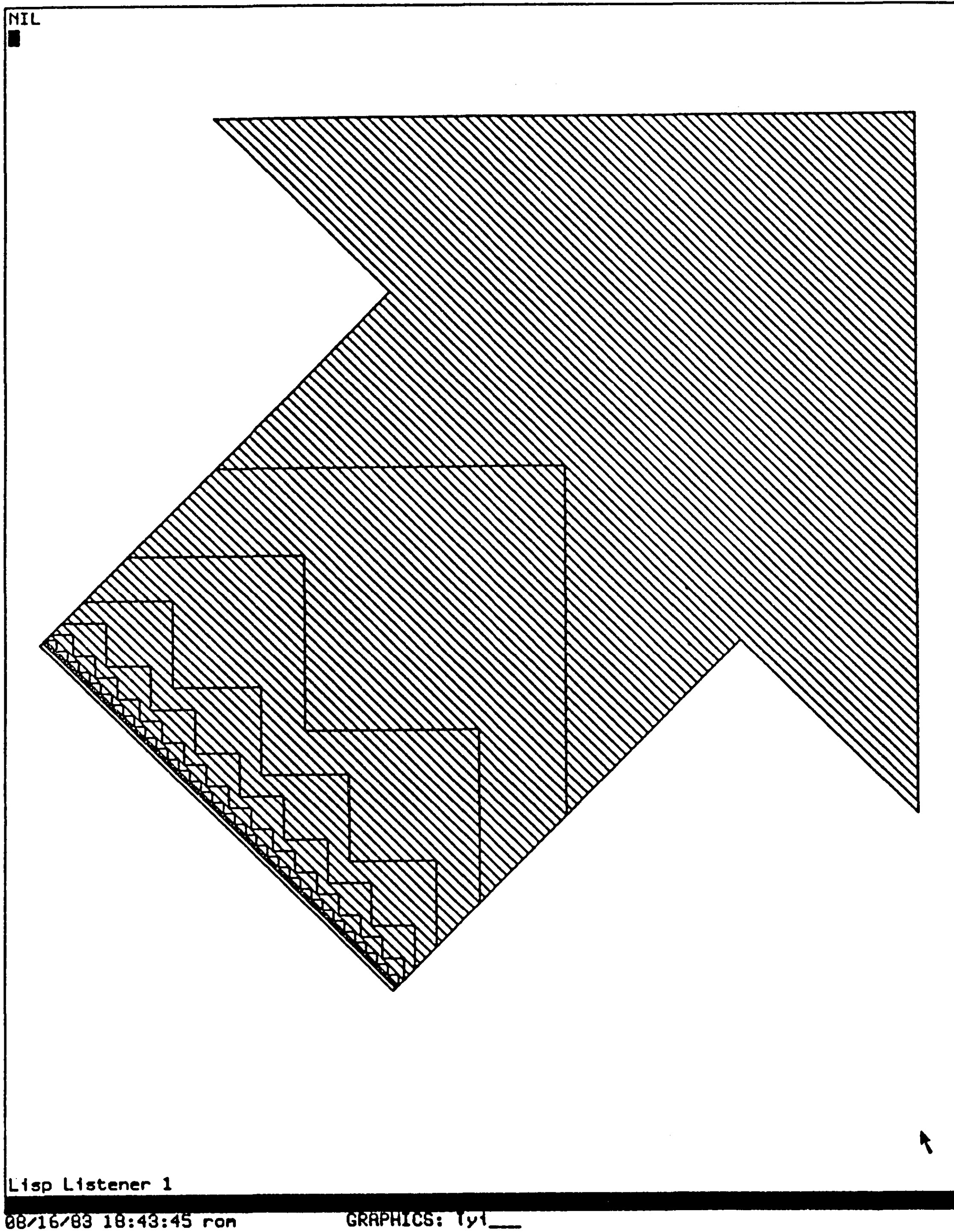


Figure 2. Program output with stripes of even spacing and density.

```
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:          ("dess" "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*.

2.6.2 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 **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 "Program Development Help Facilities".

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 "Functions: Program Development Tools and Techniques", page 40.

Reference

(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.

(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.

2.6.3 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 "Functions: Program Development Tools and Techniques", page 40.

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: /dess/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 **defconst**.

2.6.4 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".

2.6.4.1 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: Program Development Tools and Techniques", page 142.

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 (m-.)
: 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.

2.6.4.2 Names

Often you know only part of a function name and need to find the complete name. Use Function Apropos (m-X).

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-X) 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 c-. or click left on names in the display to edit the definitions of the functions listed.

2.6.4.3 Documentation

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, m-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.

2.6.4.4 Argument Lists

Quick Arglist (c-sh-A) and **arglist** retrieve the argument list for a function. 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 *Reference Guide to Symbolics-Lisp*. See the section "Program Development Help Facilities".

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-A:

```
DRAW-ARROW-GRAPHIC: (*TOP-EDGE* *POX* *POY*)
```

Reference

Quick Arglist (c-sh-A)	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.

2.6.4.5 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 (m-X)	Lists functions that call the specified function. Press c-. to edit the definitions of the functions listed.
Multiple List Callers (m-X)	Lists functions that call the specified functions. Continues prompting for function names until you press only RETURN. Press c-. to edit the definitions of the functions listed.
Edit Callers (m-X)	Prepares for editing the definitions of functions that call the specified function. Press c-. to edit subsequent definitions.
Multiple Edit Callers (m-X)	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.

2.6.5 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 (c-X c-D)	Displays the current buffer's file's directory. With an argument of c-U, prompts for a directory to display.
View Directory (m-X)	Lists a directory.
↑r Dired (c-X D)	Edits the current buffer's file's directory. With an argument of c-U, prompts for a directory to edit. Displays the files in the directory. You can use single-character commands to operate on the files.
Dired (m-X)	Edits a directory. Displays the files in the directory. You can use single-character commands to operate on the files.

2.7 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 ((nl (fix (* *stripe-d* (compute-dens x)))))
|     ;; Supply at least one line
|     (cond ((≤ nl 1) 1)
|           ;; But leave some white space between lines
|           ((≥ nl (- *stripe-d* 1)) (- *stripe-d* 2))
|           (t nl))))
```

```
;;; 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 3 shows the output of the program with stripes of varying spacing and thickness.

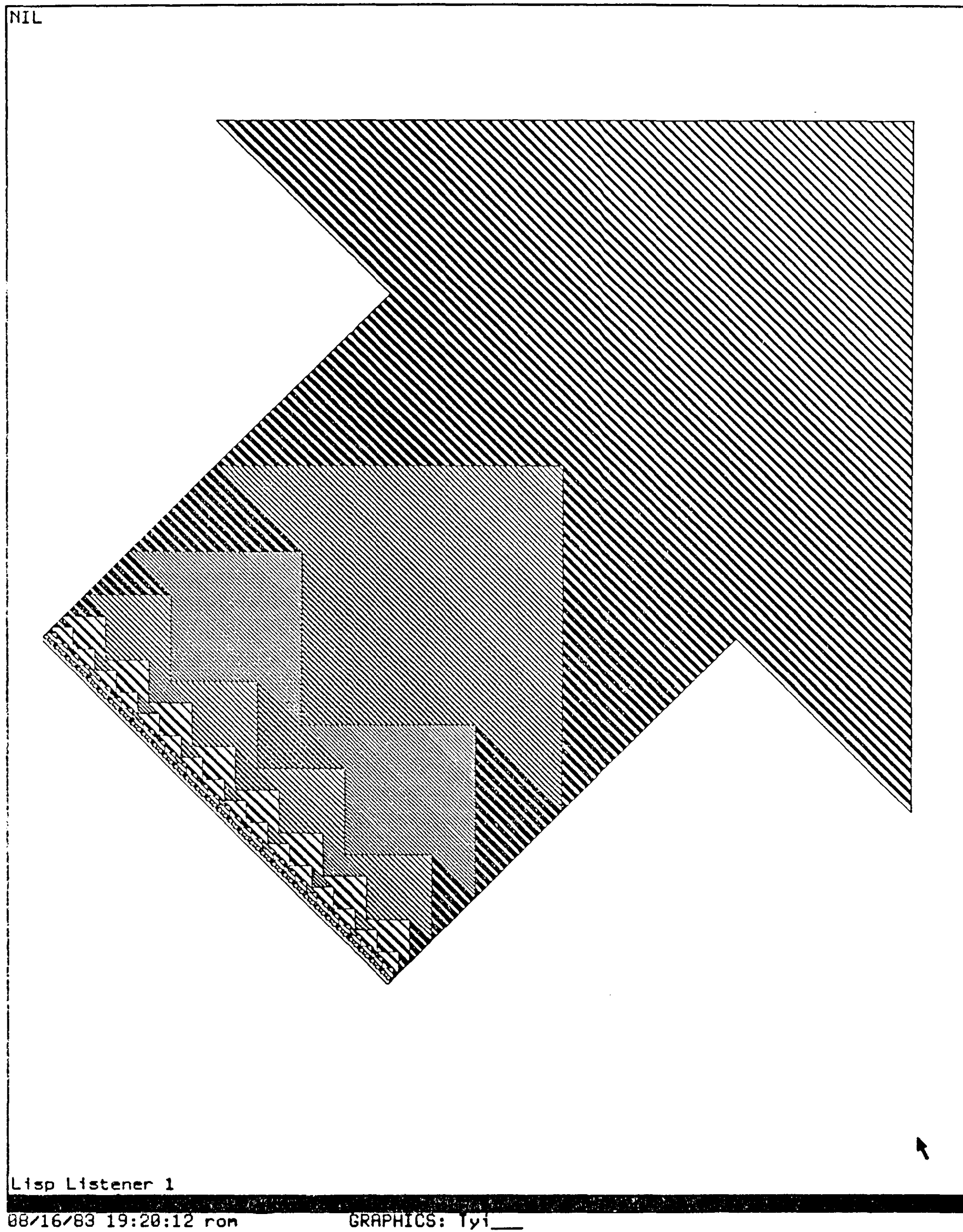


Figure 3. Program output with stripes of varying spacing and density.

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 35. See the section "Editing Code: Program Development Tools and Techniques", page 56.

2.8 Editing Code

Some features are useful mainly in composing new code. See the section "Getting Started: Program Development Tools and Techniques", page 9. See the section "Keeping Track of Lisp Syntax", page 23. Other features are helpful in both writing and editing code. See the section "Finding Out About Existing Code", page 35. In this section we discuss features that are likely to be most useful in editing existing code.

2.8.1 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 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.

2.8.2 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 35. 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 187.

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-%) 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**.

2.8.3 Moving Text

2.8.3.1 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 (c-X B)	Moves to another buffer, reading the buffer name from the minibuffer. With a numeric argument, creates a new buffer.
Select Previous Buffer (c-m-L)	Moves to the previously selected buffer.
Save Position (c-X S)	Stores the position of point in a register. Prompts for a register name.
Jump To Saved Position (c-X J)	Moves point to a position stored in a register. Prompts for a register name.
Set Pop Mark (c-SPACE)	With no argument, sets the mark at point and pushes point onto the mark stack. With an argument of c-U, pops the mark stack.

Push Pop Point Explicit (m-SPACE) With no argument, pushes point onto the mark stack without setting the mark. With an argument *n*, exchanges point with the *n*th position on the mark stack.

Move To Previous Point (c-m-SPACE) Exchanges point and the top of the mark stack.

Swap Point And Mark (c-X c-X) Exchanges point and mark. Activates the region between point and mark. Use Beep (c-G) to turn off the region.

2.8.3.2 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 (c-m-K)	Kills forward one or more Lisp expressions.
Backward Kill Sexp (c-m-RUBOUT)	Kills backward one or more Lisp expressions.
Mark Definition (c-m-H)	Puts point and mark around the current definition.
Save Region (m-W)	Pushes the text of the region onto the kill ring without killing the text.
Yank (c-Y)	Pops the last killed text from the kill ring, inserting the text into the buffer at point. With an argument *n*, yanks the *n*th entry in the kill ring. Does not rotate the kill ring.
Yank Pop (m-Y)	After a c-Y 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.
[*Region* (M2)]	When *region* is defined, pushes the text of *region* onto the kill ring without killing the text (like m-W). Repeated execution has the following effects: • First repetition: kills the text of *region*, pushing the text onto the kill ring (like c-W)

- 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).

2.8.3.3 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 (c-X X)	Copies contents of the region to a register. Prompts for a register name.
Open Get Register (c-X G)	Inserts contents of a register into the current buffer at point. Prompts for a register name.

2.8.3.4 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 (c-X H)	Marks an entire buffer.
Insert Buffer (m-X)	Inserts contents of the specified buffer into the current buffer at point.
Insert File (m-X)	Inserts contents of the specified file into the current buffer at point.

2.8.4 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 (c-X ()	Begins recording keystrokes as a keyboard macro.
End Kbd Macro (c-X))	Stops recording keystrokes as a keyboard macro.
Call Last Kbd Macro (c-X E)	Executes the last keyboard macro.
Name Last Kbd Macro (m-X)	Gives the last keyboard macro a name.
Install Macro (m-X)	Installs on a key the last keyboard macro or a named macro.
Install Mouse Macro (m-X)	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.

2.8.5 Using Multiple Windows

2.8.5.1 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 "The Compiler Warnings Database: Program Development Tools and Techniques", page 79.

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: Program Development Tools and Techniques", page 142.

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 O)	Moves to the other of two windows.
Scroll Other Window (c-m-V)	Scrolls the other of two windows.
One Window (c-X 1)	Returns to one-window display, selecting the buffer the cursor is in.

2.8.5.2 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 4 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.

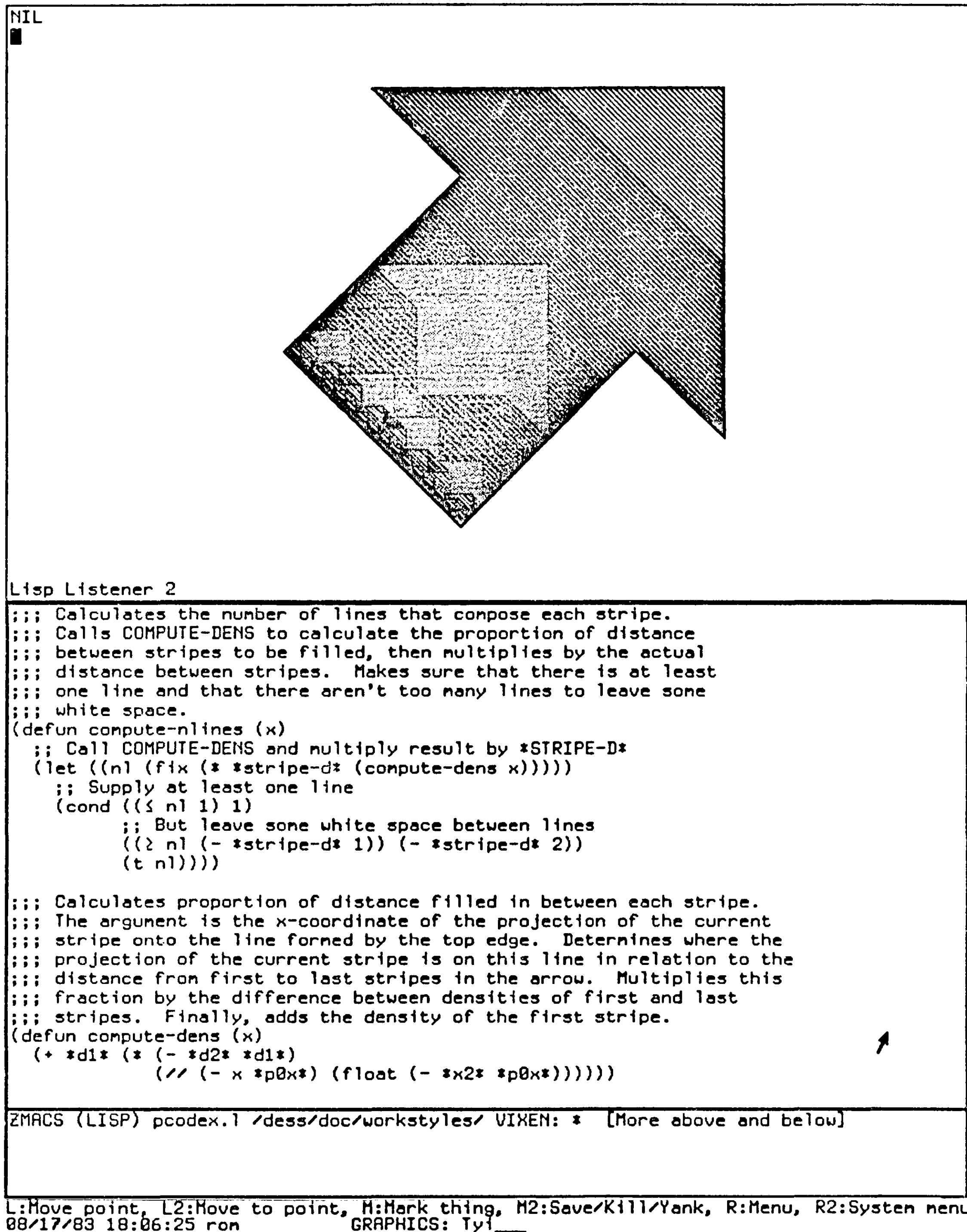


Figure 4. Using multiple windows to test the program while viewing the source code.

2.8.5.3 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*. See the section "Window System Choice Facilities" in *Programming the User Interface*. For examples of simple uses of windows, including choose-variable-values windows: See the section "Using Flavors and Windows", page 111.

3. 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, named-lambda**, **subst**, or **named-subst**. These functions are executed by the Lisp evaluator. The most common interpreted functions you define are **named-lambdas**. When you load a source file that contains **defun** forms or when you otherwise evaluate these forms, you create **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 Lisp Machine 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 70.
- 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 73.

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 *Reference Guide to Symbolics-Lisp*.

3.1 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 **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 35.

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 79.

For more information on the compiler: See the section "The Compiler", page 305.

3.1.1 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:

Form	*Action*
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 **defun**, 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 **macro**, 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 **defmacro** form expands into this kind of form.
Special case	Some forms, like **eval-when**, **declare**, and **progn 'compile** 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".
Atom, **comment** 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.

3.1.2 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 **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 **:bin**.

(compiler:compile-file *file-name***)**
: Compiles a file, placing the compiled code in a file of canonical type **:bin**.

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.

3.2 Evaluating Lisp Code

3.2.1 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 **step** or the **:step** option for **trace**. See the section "Tracing and Stepping: Program Development Tools and Techniques", page 92. 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 **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: Program Development Tools and Techniques", page 94.

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) [*Zmacs Window* (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.

3.2.2 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 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.

`m-Y` — 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*.

4. Debugging Lisp Programs

The Symbolics computer offers a variety of tools for debugging Lisp programs. The kind of debugging aid you use depends on the application of the program. Bugs might be more obvious in a graphics program 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 aids 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.

4.1 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", page 222. 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 70. Figure 5 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.

4.2 The Debugger

Some errors during execution automatically invoke the Lisp Machine's Debugger. You can enter the Debugger at other times by pressing c-m-SUSPEND. You can also enter the Debugger from within a program by inserting a call to **dbg** (with no arguments) into the code and recompiling. You can force a process into the Debugger by calling **dbg** with an argument of *process*. See the section "Breakpoints: Program Development Tools and Techniques", page 98.

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.

A window-oriented version of the Debugger is the Display Debugger. Invoke it from within the Debugger by pressing c-m-W.

Example

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

```
Warnings for file VIXEN: /dess/doc/workstyles/pcodex.‡

█ 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 /dess/doc/workstyles/ VIXEN:
ZMACS (LISP) pcodex.l /dess/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 5. Edit Compiler Warnings (m-X) splits the screen. The upper window contains compiler warnings. The lower window contains the source code.

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 *Reference Guide to Symbolics-Lisp*. We can use one of these options, or we can use other Debugger commands to examine or manipulate the stack. Let's use `c-m-W` to invoke the Display Debugger.

Figure 6 shows the Display 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 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 7 shows the Display 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 Display Debugger by pressing c-Z. We can then set the value of ***x2*** and reinvoke the frame.

In the Debugger, c-HELP displays information on Debugger commands. Following are some of the most useful commands:

Reference

c-A	Shows arguments for the current stack frame.
c-E	Calls the editor to edit the function from the current frame.
c-L	Clears the screen and redisplays the original error message.
c-N	Goes down the stack by one frame.
c-P	Goes up the stack by one frame.
c-R	Returns a value from the current frame.
m-B	Shows a backtrace of function names with arguments.
m-L	Shows local variables and disassembled code for the current frame.
c-m-R	Reinvokes the current frame.
c-m-W	Invokes the Display Debugger.

4.3 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: Program Development Tools and Techniques", page 23.

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**:

```
                                   More above
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 *X2*
    11 PUSH-INDIRECT *P0X*
=>  12 BUILTIN --INTERNAL STACK
    13 BUILTIN FLOAT STACK
    14 BUILTIN /-INTERNAL STACK
                                   More below

  #<Stack-Frame COMPUTE-DENS PC=12>

Args:                              Locals:
Arg 0 (X): 1800

                                   More above
   (DO-ARROW)
   (DRAW-ARROW-GRAPHIC 1280 1800 1800)
   (DRAW-BIG-ARROW)
   (STRIPE-ARROWHEAD)
   (COMPUTE-NLINES 1800)
  →(COMPUTE-DENS 1800)
                                   More below

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

 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:       Tyi
```

Figure 6. The Display Debugger: inspecting the stack frame containing a call to **compute-dens**.

```
                                    Top of object
*X2*
Value is NIL
Function is unbound
Property list: (DOCUMENTATION "..." SPECIAL #<UNIX-PATHNAME "VIXEN: //dess//workstyles.
Package: #<Package GRAPHICS 36635277>

                                    Bottom of object
   #<Stack-Frame COMPUTE-DENS PC=12>
   *X2*

Args:                                      Locals:
Arg 0 (X): 1000

                                    More above
   (DO-ARROW)
   (DRAW-ARROW-GRAPHIC 1280 1000 1000)
   (DRAW-BIG-ARROW)
   (STRIPE-ARROWHEAD)
   (COMPUTE-NLINES 1000)
  →(COMPUTE-DENS 1000)
                                    More below
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

 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 17:02:05 rom                     GRAPHICS:      Tyi
```

Figure 7. The Display Debugger: inspecting the variable ***x2***.

```
(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 8. 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 9. 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 10. 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 1.

Example

Figure 4 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:

```
(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))
```

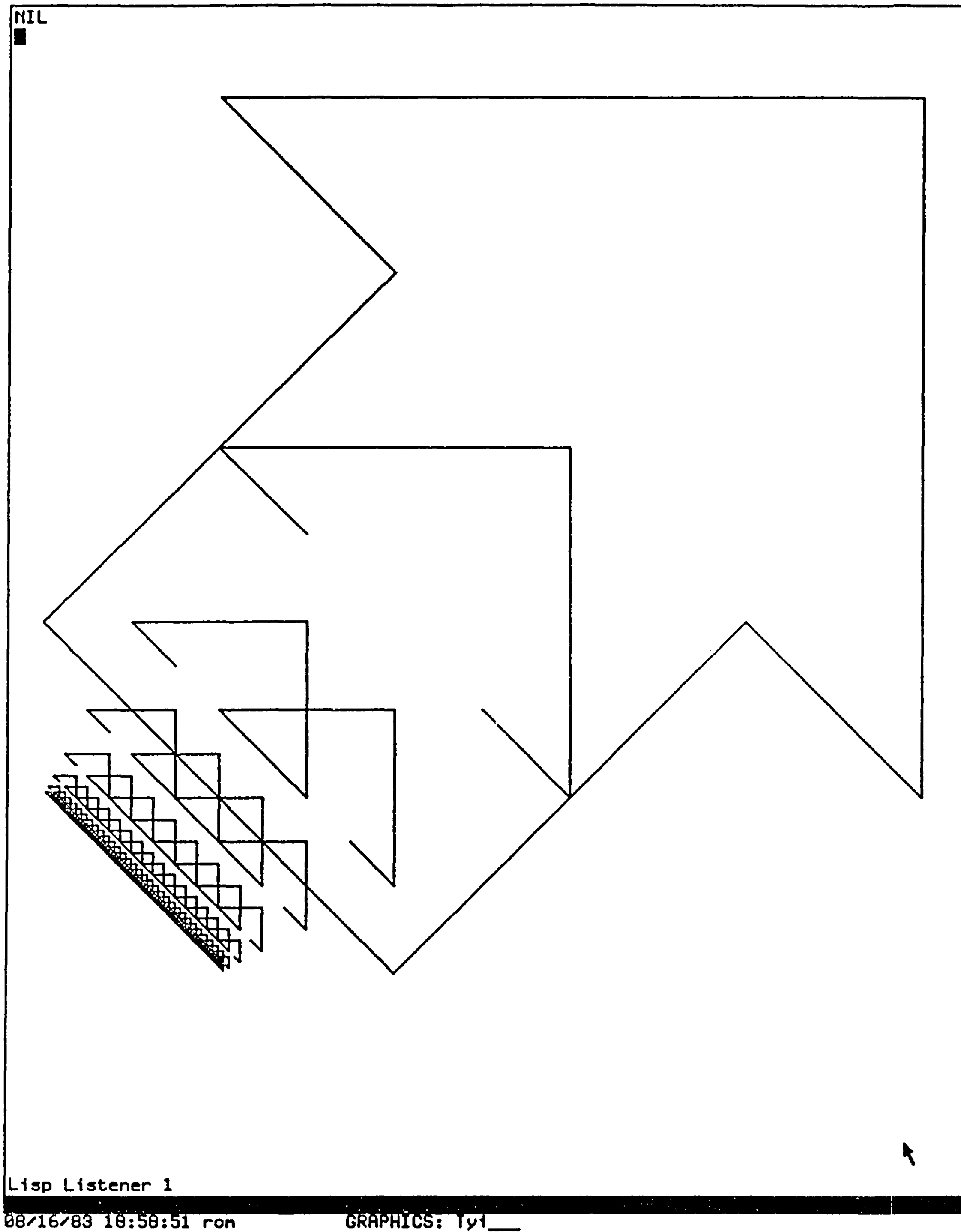


Figure 8. Output resulting from a faulty attempt to outline the small arrows recursively.

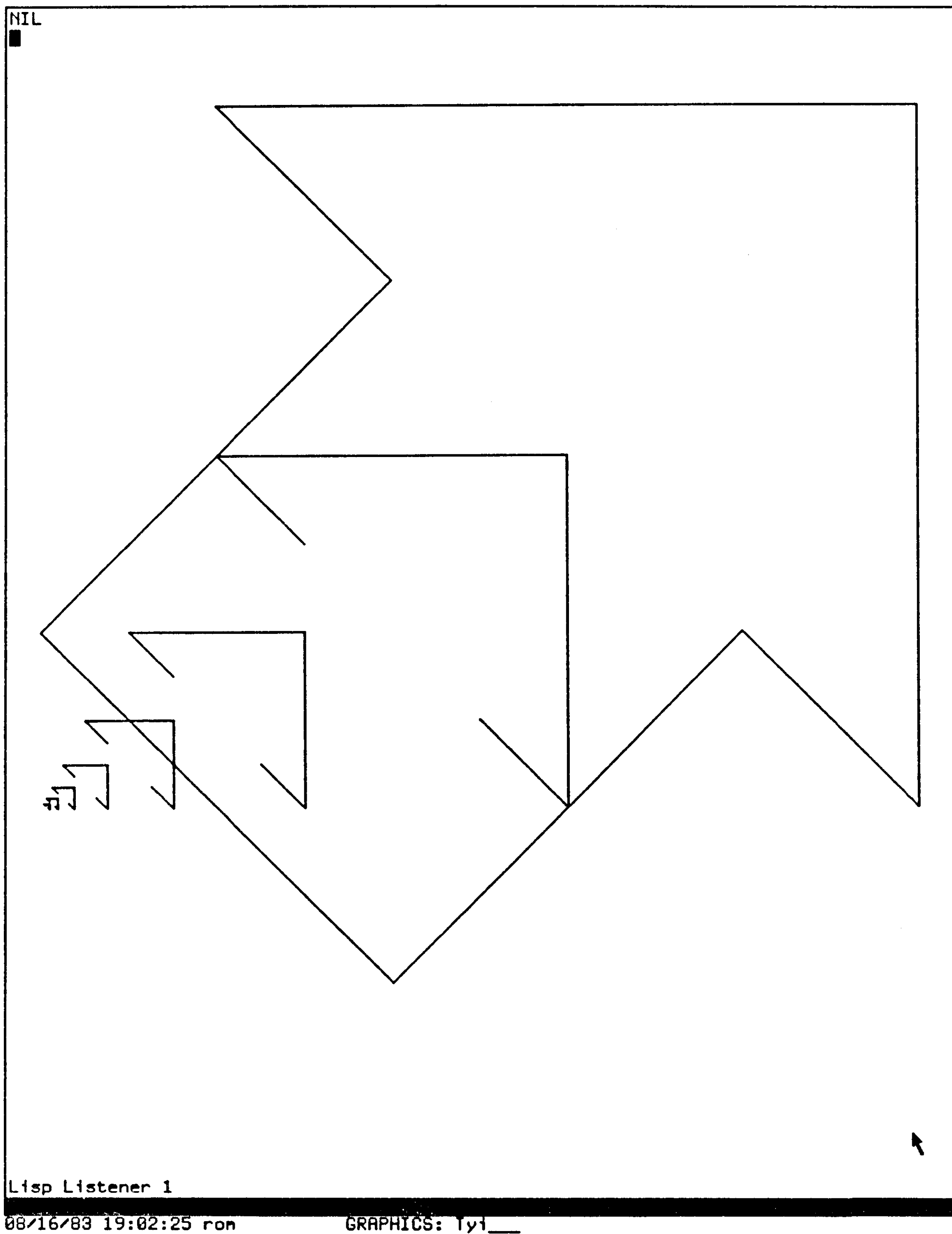


Figure 9. Output resulting from a faulty attempt to outline the small arrows recursively, with the second function call commented out.

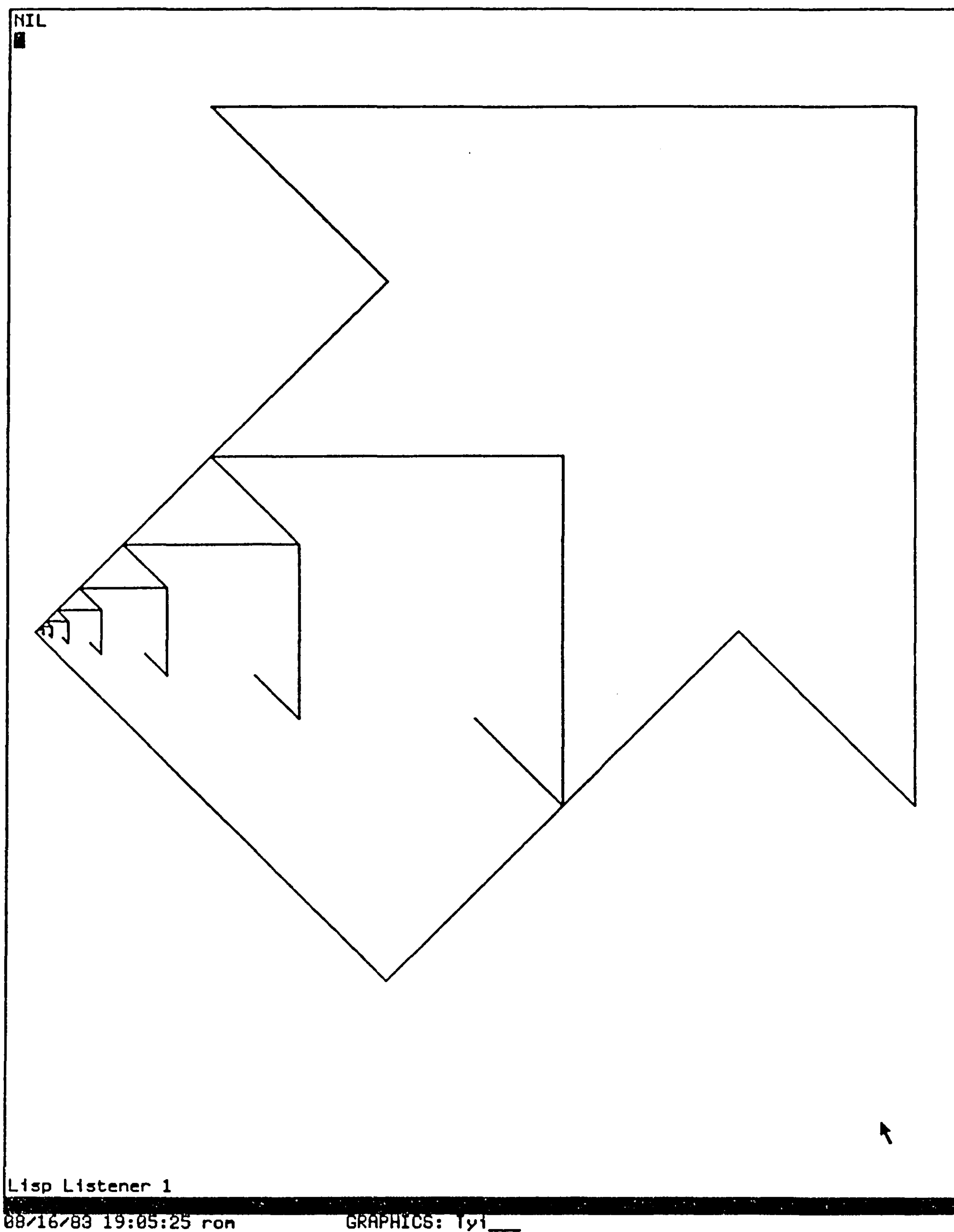


Figure 10. Output resulting from a corrected attempt to outline the small arrows recursively, with the second function call commented out.

4.4 Tracing and Stepping

4.4.1 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:

- Click on [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 276.

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.

4.4.2 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 **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 75.

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 (⇄) 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

Table 1. Trace Menu Items and **trace** Options

Trace menu item	**trace** *option*	*Description*
[Cond break before]	**:break** *predicate*	Enters breakpoint on function entry if *predicate* not **nil**
[Break before]	**:break t**	Enters breakpoint on function entry
[Cond break after]	**:exitbreak** *predicate*	Enters breakpoint on function exit if *predicate* not **nil**
[Break after]	**:exitbreak t**	Enters breakpoint on function exit
[Error]	**:error**	Enters Debugger on function entry
[Step]	**:step**	Steps through (interpreted) function execution
[Cond before]	**:entrycond** *predicate*	Prints trace output on function entry if *predicate* not **nil**
[Cond after]	**:exitcond** *predicate*	Prints trace output on function exit if *predicate* not **nil**
[Conditional]	**:cond** *predicate*	Prints trace output on function entry and exit if *predicate* not **nil**
[Print before]	**:entryprint** *form*	Prints value of *form* in trace entry output
[Print after]	**:exitprint** *form*	Prints value of *form* in trace exit output
[Print]	**:print** *form*	Prints value of *form* in trace entry and exit output
[ARGPDL]	**:argpdl** *pdl*	On function entry, pushes list of function name and args onto *pdl*; pops list on function exit
[Wherein]	**:wherein** *function*	Traces function only when called within *function*
[Per Process]	**:per-process** *process*	Traces function only in *process*
[Untrace]		Calls **untrace** on function
	:entry *list*	Prints values of forms in *list* on function entry
	:exit *list*	Prints values of forms in *list* on function exit
	:arg :value :both :nil	Controls printing of args on function entry and values on function exit

breakpoint loop. For a list of commands, press HELP inside the Stepper, or: See the section "Stepping Through an Evaluation", page 287. Following are some basic Stepper commands:

Command	*Action*
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: Program Development Tools and Techniques", page 92. 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

(**step** *form*) — Steps through the evaluation of *form*

Trace (m-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** *function-spec* **:step**)) — Steps through the execution of a function being traced. If *function-spec* is a symbol, the keyword **:function** is unnecessary.

4.5 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 **dbg** (with no arguments) or to **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 *Reference Guide to Symbolics-Lisp*. 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

(**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.

(**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.

4.6 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 11. 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 **loop** form. We move to the beginning of the **loop** form and expand it, using c-sh-M:

```
((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)
```

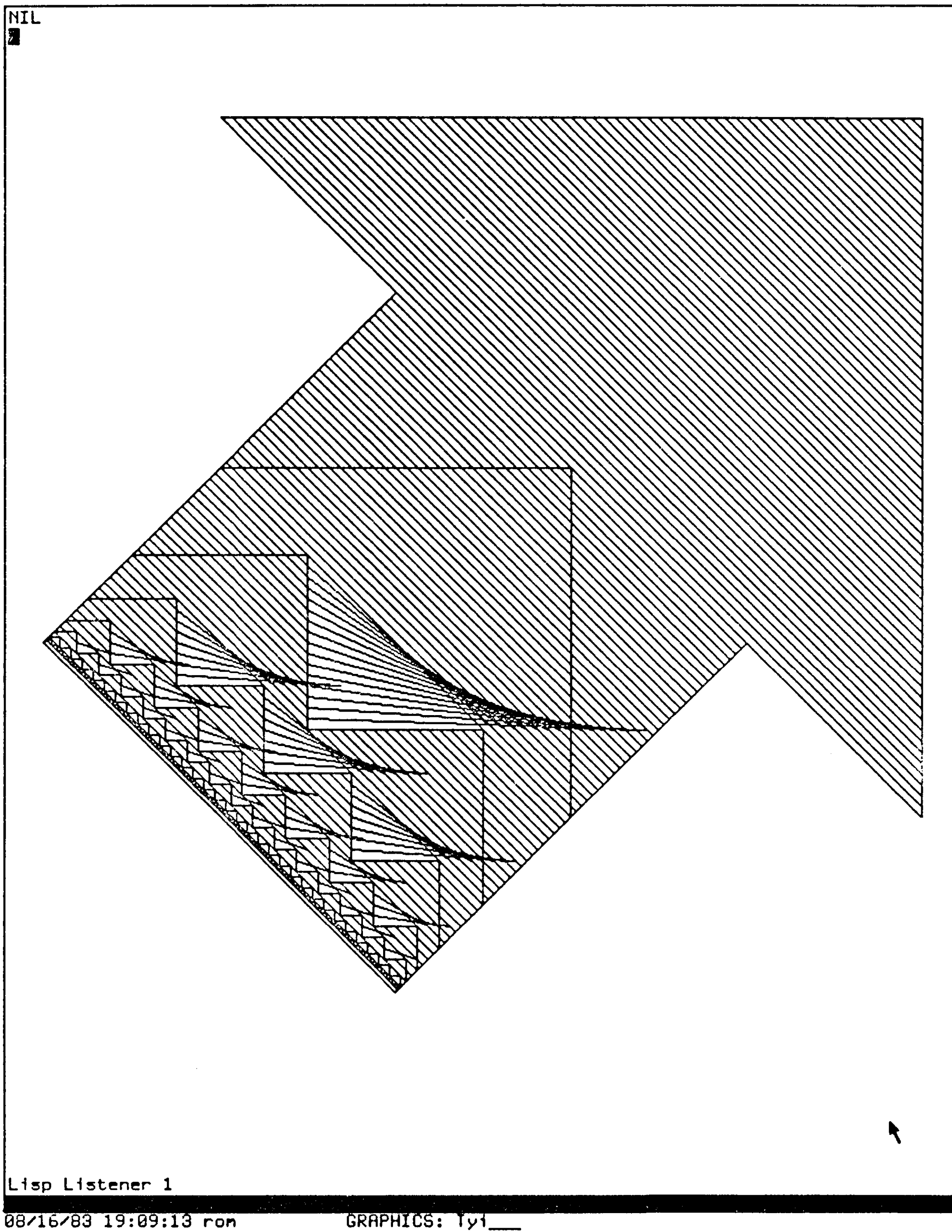


Figure 11. Output from the program with a bug in the function **draw-arrow-shaft-stripes**.

The expansion shows the **lambda**-bindings and **prog** form that the **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 **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 **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**.

4.7 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 293.

Example

Suppose we had represented each arrow as an instance of a structure (defined with **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 12 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 13 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

(inspect *object***)**	Selects an Inspector window in which to inspect *object*.
`SELECT I`	Selects an Inspector window.
[Inspect] (from the System menu)	Selects an Inspector window.
(disassemble *function***)**	Prints a representation of the assembly-language instructions for a compiled function.
Disassemble (`m-X`)	Prompts for the name of a compiled function and displays a

```
*arr*

                                Top of History                          Exit
   #<ARROW -33247021>                                                   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 -33247021>
Named structure of type 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.
08/17/83 18:23:32 ron          GRAPHICS: Tyi___
```

Figure 12. The Inspector window: inspecting an instance of a structure.

representation of the function's assembly-language instructions.

```
*arr*

                    Top of History                          Exit
  #<ARROW 10020042>                                         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 ron                GRAPHICS:      Tyi
```

Figure 13. The Inspector window: inspecting an instance of a flavor.

```
*arr*

                              Top of History                          Exit
    #<ARROW 10020042>                                                 Return
    #<Message handler for ARROW>                                      Modify
                                                                      DeCache
                                                                      Clear
                                                                      Set \
                              Bottom of History
                              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
                              More below
                              Top of object
#<Message handler for ARROW>
:COMPUTE-DENS:                 #'(:METHOD BASIC-ARROW :COMPUTE-DENS)
:COMPUTE-NLINES:               #'(:METHOD BASIC-ARROW :COMPUTE-NLINES)
:COMPUTE-POINTS:               #'(:METHOD BASIC-ARROW :COMPUTE-POINTS)
:COMPUTE-STRIPE-D:             #'(:METHOD BASIC-ARROW :COMPUTE-STRIPE-D)
:COMPUTE-TOP-EDGES:            #'(:METHOD BASIC-ARROW :COMPUTE-TOP-EDGES)
DESCRIBE:                      #'(:METHOD SI:VANILLA-FLAVOR DESCRIBE)
:DRAW-ARROW:                   #'(:METHOD BASIC-ARROW :DRAW-ARROW)
:DRAW-ARROW-SHAFT-LINES:       #'(:METHOD ARROW-MIXIN :DRAW-ARROW-SHAFT-LINES)
:DRAW-ARROW-SHAFT-STRIPES:     #'(:METHOD ARROW-MIXIN :DRAW-ARROW-SHAFT-STRIPES)
:DRAW-ARROWHEAD-LINES:         #'(:METHOD BASIC-ARROW :DRAW-ARROWHEAD-LINES)
:DRAW-OUTLINE:                 #'(:METHOD ARROW-MIXIN :DRAW-OUTLINE)
:EVAL-INSIDE-YOURSELF:         #'(:METHOD SI:VANILLA-FLAVOR :EVAL-INSIDE-YOURSELF)
:FUNCALL-INSIDE-YOURSELF:      #'(:METHOD SI:VANILLA-FLAVOR :FUNCALL-INSIDE-YOURSELF)
GET-HANDLER-FOR:               #'(:METHOD SI:VANILLA-FLAVOR GET-HANDLER-FOR)
:OPERATION-HANDLED-P:          #'(:METHOD SI:VANILLA-FLAVOR :OPERATION-HANDLED-P)
:P0X:                          #'(:METHOD BASIC-ARROW :P0X)
:P0Y:                          #'(:METHOD BASIC-ARROW :P0Y)
:PRINT-SELF:                   #'(:METHOD SI:VANILLA-FLAVOR :PRINT-SELF)
:SEND-IF-HANDLES:              #'(:METHOD SI:VANILLA-FLAVOR :SEND-IF-HANDLES)
:SET-STRIPE-D:                 #'(:METHOD BASIC-ARROW :SET-STRIPE-D)
:STRIPE-ARROW-SHAFT:           #'(:METHOD ARROW-MIXIN :STRIPE-ARROW-SHAFT)
:STRIPE-ARROWHEAD:             #'(:METHOD BASIC-ARROW :STRIPE-ARROWHEAD)

                              More below
Choose a value by pointing at the value.  Right finds function definition.
08/20/83 17:09:42 rom                      GRAPHICS:      Tyi
```

Figure 13, *continued.*

```
*arr*

                              Top of History                          Exit
   #<ARROW 10020042>                                                  Return
   #<Message handler for ARROW>                                       Modify
   #'(:METHOD BASIC-ARROW :COMPUTE-DENS)                              DeCache
                                                                      Clear
                                                                      Set \
                              Bottom of History
                              Top of object
#<ARROW 10020042>
An instance of ARROW.  #<Message handler for ARROW>

DEPTH:        6
TOP-EDGE:     10
TOP-EDGE-2:   5
                              More below
                              Top of object
#<Message handler for ARROW>
:COMPUTE-DENS:              #'(:METHOD BASIC-ARROW :COMPUTE-DENS)
:COMPUTE-NLINES:            #'(:METHOD BASIC-ARROW :COMPUTE-NLINES)
:COMPUTE-POINTS:            #'(:METHOD BASIC-ARROW :COMPUTE-POINTS)
:COMPUTE-STRIPE-D:          #'(:METHOD BASIC-ARROW :COMPUTE-STRIPE-D)
:COMPUTE-TOP-EDGES:         #'(:METHOD BASIC-ARROW :COMPUTE-TOP-EDGES)
                              Top of object
#<DTP-COMPILED-FUNCTION (:METHOD BASIC-ARROW :COMPUTE-DENS) 46660073>
     0 ENTRY: 4 REQUIRED, 0 OPTIONAL
     1 PUSH-INDIRECT *D1*
     2 PUSH-INDIRECT *D2*
     3 PUSH-INDIRECT *D1*
     4 BUILTIN --INTERNAL STACK
     5 PUSH-LOCAL FP|3         ;X
     6 PUSH-INSTANCE-VARIABLE 2 ;P0X
     7 BUILTIN --INTERNAL STACK
    10 PUSH-INSTANCE-VARIABLE 15        ;X2
    11 PUSH-INSTANCE-VARIABLE 2 ;P0X
    12 BUILTIN --INTERNAL STACK
    13 BUILTIN FLOAT STACK
    14 BUILTIN /-INTERNAL STACK
    15 BUILTIN *-INTERNAL STACK
    16 BUILTIN +-INTERNAL STACK
    17 RETURN-STACK

                              Bottom of object
Choose a value by pointing at the value.  Right finds function definition.
08/20/83 17:10:06 rom                    GRAPHICS:      Tyi
```

Figure 13, *concluded.*

5. Using Flavors and Windows

All Lisp Machine Lisp 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 *Reference Guide to Symbolics-Lisp.*
- On windows: See the section "Using the Window System" in *Programming the User Interface.*
- On menus: See the section "Window System Choice Facilities" in *Programming the User Interface.*
- On conditions and errors: See the section "Conditions" in *Reference Guide to Symbolics-Lisp.*

5.1 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 **terminal-io** is bound:

```
(defflavor 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 redisplays 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.

5.1.1 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.

```
(defflavor 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))))))
```

5.1.2 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 *Reference Guide to Symbolics-Lisp*.)

```
;;; 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))
```

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 **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 *Reference Guide to Symbolics-Lisp*.
- 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 *Reference Guide to Symbolics-Lisp*.

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.

5.1.3 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."))
```

5.1.4 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**.

```
(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."))
```

5.1.5 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))))
```

```
(defun do-arrow ()
  (tv:choose-variable-values
        .
        .
  ;; If figure is infinitely small, just return nil
  (cond ((= *fill-proportion* 0) nil)
        ;; If screen output, make a window
        ((equal *dest-string* "screen")
         (tv:make-window 'arrow-window
                         ;; Initialize instance variables to
                         ;; values set by the user
                         ':do-stripes *do-the-stripes*
                         ':max-dep *max-depth*
                         ':fill-prop *fill-proportion*))))
```

If the output destination is "LGP" or "File", we want to make an instance of **lgp-pixel-stream** with the instance variable **stream** initialized to an appropriate stream. We construct this stream by calling **si:make-hardcopy-stream** with an argument that depends on the output destination. We use **with-open-stream** to produce the output on the stream and close it when we finish.

```
(defun do-arrow ()
  (tv:choose-variable-values
        .
        .
  (cond ((= *fill-proportion* 0) nil)
        ;; If screen output, make a window
        ((equal *dest-string* "screen")
         (tv:make-window 'arrow-window
                         ;; Initialize instance variables to
                         ;; values set by the user
                         ':do-stripes *do-the-stripes*
                         ':max-dep *max-depth*
                         ':fill-prop *fill-proportion*))
        ;; If LGP or file output, use an appropriate stream
        (t (with-open-stream
             (stream
               ;; This function returns a stream suitable for
               ;; LGP output
               (si:make-hardcopy-stream
                 ;; Argument is the output device.  For LGP,
                 ;; use the default hardcopy device.
                 (if (equal *dest-string* "lgp")
                     si:*default-hardcopy-device*
                     ;; For file output, use the correct format
                     ;; for the hardcopy device and direct
                     ;; output to the file specified by the user
                     (lgp:get-lgp-record-file-hardcopy-device
                       *output-file*))))
             ;; Make an instance of our LGP output flavor
             (let ((*dest*
                     (make-instance 'lgp-pixel-stream
                                    ;; Initialize instance
                                    ;; variable to output stream
                                    ':output-stream stream)))
               ;; Position the figure on the page
               (send *dest* ':compute-parameters)
               ;; Draw the figure, using instance-variable values
               ;; as arguments
               (draw-arrow-graphic (send *dest* ':top-edge)
                                   (send *dest* ':right-x)
                                   (send *dest* ':top-y)))))))
```

5.1.6 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 ...))

(defun do-arrow ()
  (tv:choose-variable-values
        .
        .
  (cond ((= *fill-proportion* 0) nil)
        ;; If screen output, make a window
        ((equal *dest-string* "screen")
         (tv:make-window 'arrow-window
                         ;; Initialize instance variables to
                         ;; values set by the user
                         ':do-stripes *do-the-stripes*
                         ':max-dep *max-depth*
                         ':fill-prop *fill-proportion*
                         ;; Specify top-level function for the
                         ;; process associated with the window
                         ':process '(window-loop)))
        .
        .
```

We next want to be able to handle mouse clicks. We include the flavors **tv:any-tyi-mixin** and **tv:list-mouse-buttons-mixin** in **basic-arrow-window**. When a window is waiting for input and the mouse is clicked while over the window, a *blip* enters the window's input buffer. A blip is a list whose form, with **tv:list-mouse-buttons-mixin**, is as follows:

(:mouse-button *encoded-click window x y***)**

Encoded-click is a fixnum that represents the button clicked.

```
(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 ...))
```

We also want a mouse documentation string to appear when the mouse is over the window:

```
(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 redisplays 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.

5.1.7 Signalling 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**.

```
(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))
```

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 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 **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 **query-io** to the window. If **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 **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)
```

5.2 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 "The Inspector: Program Development Tools and Techniques", page 104. 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*. 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: Program Development Tools and Techniques", page 142.

5.2.1 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 **typep**. Given an instance and a flavor name, **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 **query-io** is bound to a window before turning on a blinker. We find out whether the object bound to **query-io** is built on **tv:sheet** by using **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.

5.2.2 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.
(send *instance* **':which-operations)**	Returns a list of messages that *instance* 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*.

5.2.3 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.

6. Calculation Module for the Sample Program

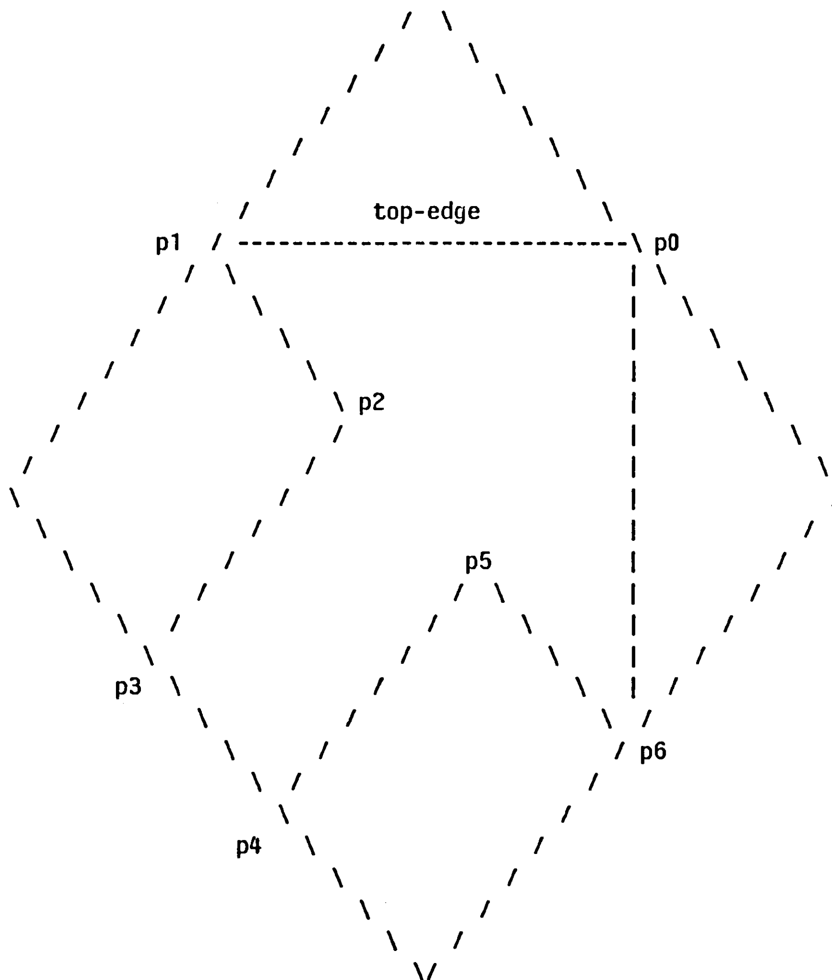
The program used as an example in this document draws the recursive arrow graphic on the document's cover. This section contains Lisp code that calculates coordinates for the endpoints of the lines that compose the figure. The code produces output by sending messages to instances of flavors defined in 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 165. For a reproduction of the LGP graphic the program produces: See the section "Graphic Output of the Sample Program", page 185.

```
;;; -*- 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 ((nl (fix (* *stripe-d* (compute-dens x)))))
    ;; Supply at least one line
    (cond ((≤ nl 1) 1)
          ;; But leave some white space between lines
          ((≥ nl (- *stripe-d* 1)) (- *stripe-d* 2))
          (t nl))))
```

```
;;; 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)))
```

7. 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 147. For a reproduction of the LGP graphic the program produces: See the section "Graphic Output of the Sample Program", page 185.

```
;;; -*- 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")
```

```
;; If figure is infinitely small, just return nil
(cond ((= *fill-proportion* 0) nil)
      ;; If screen output, make a window
      ((equal *dest-string* "Screen")
       (tv:make-window 'arrow-window
                       ;; Initialize instance variables to
                       ;; values set by the user
                       ':do-stripes *do-the-stripes*
                       ':max-dep *max-depth*
                       ':fill-prop *fill-proportion*
                       ;; Specify top-level function for the
                       ;; process associated with the window
                       ':process '(window-loop)))
      ;; If LGP or file output, use an appropriate stream
      (t (with-open-stream
           (stream
             ;; This function returns a stream suitable for
             ;; LGP output
             (si:make-hardcopy-stream
               ;; Argument is the output device.  For LGP,
               ;; use the default hardcopy device.
               (if (equal *dest-string* "lgp")
                   si:*default-hardcopy-device*
                   ;; For file output, use the correct format
                   ;; for the hardcopy device and direct
                   ;; output to the file specified by the user
                   (lgp:get-lgp-record-file-hardcopy-device
                     *output-file*))))
           ;; Make an instance of our LGP output flavor
           (let ((*dest*
                   (make-instance 'lgp-pixel-stream
                                  ;; Initialize instance
                                  ;; variable to output stream
                                  ':output-stream stream)))
             ;; Position the figure on the page
             (send *dest* ':compute-parameters)
             ;; Draw the figure, using instance-variable values
             ;; as arguments
             (draw-arrow-graphic (send *dest* ':top-edge)
                                 (send *dest* ':right-x)
                                 (send *dest* ':top-y)))))))
```

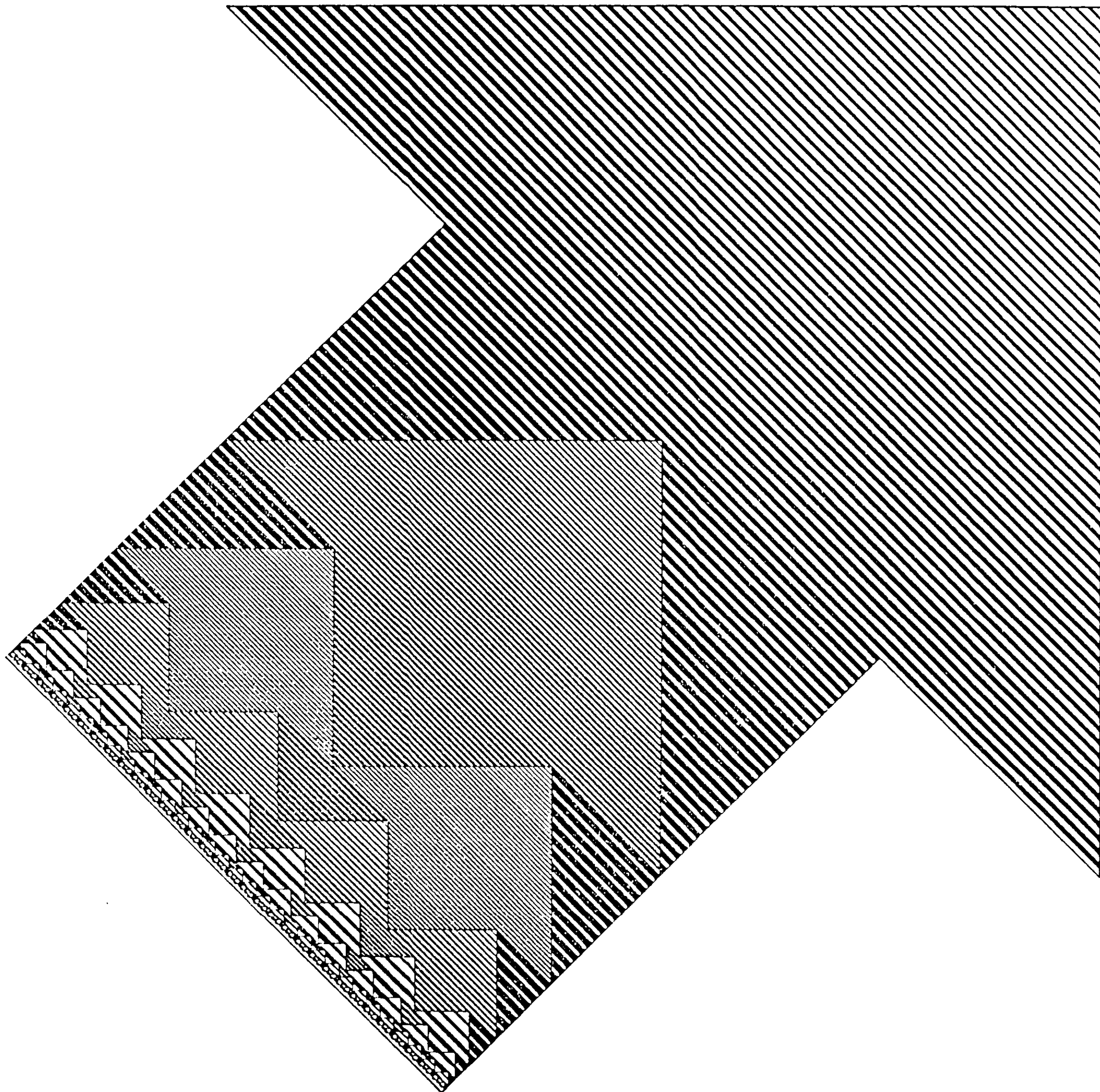
```
;;; Top-level function for process associated with arrow window.
;;; The function is called when the window is created.  Argument is
;;; the window.  The function sends the window a :MAIN-LOOP message.
;;; This method should be the actual command loop for the process.
(defun window-loop (window)
  (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 (≥ 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))


;;; 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)
```

8. 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 147. 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 165.



PART II.

Maintaining Large Programs

9. Introduction to the System Facility

When a program gets 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 file called "defs" that contains some macro definitions (or flavor or structure 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 a nearly impossible to maintain manually.

This chapter describes the *system facility*, which addresses these difficulties. 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.

1. You specify these files and relationships by *defining* the system, using the system facility's **defsystem** special form. The definition, often called a *system declaration*, specifies such information as the names of the files (or modules) in your system and what operations should be performed on which 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 191.

2. This system definition should be placed into its own file. 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!) See the section "Loading the System Definition", page 217.

3. You can *make* the system, that is, have the Symbolics Lisp Machine perform the operations specified in your system definition, by using the **make-system** function or relevant Command Processor commands (Load System and Compile System). For example, **make-system** loads all the files of the system, recompiles all the files that need compiling, and so on. See the section "Making a System", page 221.

4. 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 231.

5. Various functions exist to help you find information about existing systems. See the section "Getting Information About a System", page 249.

10. 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 system definition (called the *system declaration*) describes these relationships and rules. Some useful, general guidelines are:

1. Use Zmacs to enter the 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) and any user-defined **defsystem** transformations. Both must precede the system declaration in the file. See the section "System Declaration File", page 218.

2. 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 and 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 217.

4. Call **make-system** to compile and load your system, as in

   ```
   (make-system 'system-name :compile :noconfirm)
   ```

   make-system uses the information in the translations file to load the system declaration file, compiling it first if necessary. Alternatively, you can also use the Command Processor's Compile System command to compile and load your system.

defsystem *name* &body *options* — *Special Form*

Defines a system called *name*. The system definition (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 the previous compilation or loading of others, and (3) the characteristics of the system, for example, the package into which the object code should be compiled. **make-system** and the relevant Command Processor commands compile and load your system in accordance with the properties specified in your system declaration.

options to **defsystem** are keywords and fall into three categories:

- Characteristics of the system, for example, its name.
- Modules — sets of files that should be compiled or loaded as a unit.
- Transformations — operations (like compilation or loading) that are performed on the system's files or modules.

General Guidelines for Using defsystem

The following example shows a simple but typical **defsystem** declaration. The purpose is to illustrate what a declaration looks like, including its major components, and to provide some important definitions and general guidelines.

```
(defsystem registrar
  ;; These options describe the characteristics of the system.
  (:name "Automatic Registration System")
  (:pathname-default "reg:reg;")

  ;; The system has 3 modules.
  (:module main "top-level")
  (:module definitions ("variables" "macros" "more-macros"))
  (:module resources "resources")

  ;; The system declares that the :compile-load
  ;; transformation should be performed on each module.
  (:compile-load definitions)
  (:compile-load resources (:fasload definitions))
  (:compile-load main (:fasload definitions resources)))
```

Note these important points.

Characteristics.

- The value of the **:name** option, "Automatic Registration System", is the user-visible name, appearing in completion tables, heralds, and so on.

- The default pathname is specified as a logical pathname. You are urged to use logical pathnames in system declarations, as they 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.

Modules.

- The **:module** option defines what files or groups of files compose the system. The definitions module consists of three files; the other two modules, of only one file each. When a module consists of more than one file, the file names must be placed in a list, as in:
  `(:module definitions ("variables" "macros" "more-macros"))`
  Module declarations must precede the transformation declarations in the **defsystem** form.

- The files that compose a single module are treated as a unit. Therefore, one file in a module cannot depend on the previous loading of another file in that module. So, if "more-macros" depends on the previously loading of "macros", then "macros" and "more-macros" cannot be placed in the same module. The ordering of files within a module specification is not significant.

- The files in a single module should share the same characteristics. Customarily the set of files grouped in a module perform some similar function, as long as the files do not depend on one another. For example, all low-level definitions (variables and macros) might be placed in the same module.

Transformations.

- A **transformation** is an operation to perform on a file or module. For example, **:fasload** is a transformation and means load the specified binary files of this module. **:compile-load** is another transformation, meaning load this module, compiling it first *under certain conditions*. For example, if the call to **make-system** specifies the **:recompile** option, then all the source files are recompiled, whether or not they are newer than their associated binary versions. A call to **make-system** with no options loads but does not compile the system's files, regardless of whether or not they need to be recompiled.

- The relationship of one **defsystem** module to another can be described as a hierarchy of dependencies. For example, modules often depend on other modules having been previously loaded. The files within a module share this characteristic: they have the same dependencies.

- A *dependency* declares that under certain circumstances all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. Consider the following transformation specification.

  ```
  (:compile-load resources (:fasload definitions))
  ```

  In this case, the **:fasload** transformation on the definitions module is a dependency of the **:compile-load** transformation on the resources module. To be more precise, the **:fasload** transformation is actually a *compile-time* dependency, meaning that it applies only when the call to **make-system** includes a compile-type option, as in `(make-system 'registrar :compile)`.

- Transformations can refer to (depend on) only those transformations that have been previously declared. So resources can depend on definitions, but not vice versa.

- According to the above declaration, **make-system** will load the main module (presumably the top-level program) last, after definitions and resources have been loaded (and perhaps compiled).

- `(:compile-load main (:fasload definitions resources))` expresses this dependency.

defsystem Options

defsystem options can be categorized as (1) characteristics of the system, (2) modules, and (3) transformations. Only the first category of options is described in this section, in the order in which options conventionally appear in the declaration. (See the section "**defsystem** Modules", page 198. See the section "**defsystem** Transformations", page 201.)

:name

:name specifies the name of the system for use in printing. This is the user-visible name appearing in completion tables, heralds, and so on.

Example: Based on the following declaration, the herald displays the name of the **registrar** system as `Automatic Registration System`.

```
(defsystem registrar
  (:name "Automatic Registration System")
  (:short-name "Registration")
  (:pathname-default "reg:reg;")
  ...)
```

:short-name

:short-name specifies an abbreviated name used in constructing disk label comments and patch file names for some file systems.

Example: Based on the following declaration, the name of the patch files of the **registrar** system are constructed from the system short name. Assuming that the logical patch directory ("reg: reg; patch;") translates to "s:>sr>registrar>patch>", then the name of the first patch file for major version #1 of **registrar** is:
`"s:>sr>registrar>patch>registration-1>registration-1-1.lisp"`.

```
(defsystem registrar
  (:name "Automatic Registration System")
  (:short-name "Registration")
  (:patchable "reg: reg; patch;")
  (:pathname-default "reg: reg;")
  ...)
```

:package

:package specifies the name of an existing package in

which all transformations are performed. A package specified in the system declaration overrides the one specified in the attribute list of the files to which the transformations are being applied. Typically, the package declaration for a system is placed in the same file as the system declaration.

Example: All the modules in **mailer** are compiled/loaded into the **mail** package.

```
(defpackage mail global 4096. ())

(defsystem mailer
  ...
  (:package "mail")
  (:module defs "defs")
  (:module mult "mult")
  ...)
```

It is sometimes necessary to selectively override the system's **:package** specification, for example, when a system module needs to use the package specified by the attribute lists of its files. This end is accomplished by specifying a different package for that module. See the section "**:module** Option for **defsystem**", page 199.

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**, the package specification will prevent name conflicts. Avoid affecting symbols in the standard Symbolics-Lisp packages.

:pathname-default

:pathname-default gives a local default within the definition of the system for strings to be parsed into pathnames. 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.

```
(:pathname-default "sys:zwei;")
```

This obviates 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, specify a pathname default for the directory storing the majority of the files.

:patchable

:patchable allows you to make patches for the system. (See the section "Patch Facility", page 231.) **:patchable** takes two optional arguments: **directory** and **patch-atom**. The first optional argument, *directory*, specifies the directory to put patch files in. Otherwise patches are stored in the directory specified by **:pathname-default**.

The second optional argument, *patch-atom*, selects a name for your patch files.

Example: The following entry in the declaration for the **registrar** system means that all patch files for system **registrar** will have names like "reg-sys-1-1.lisp" or "reg-sys-1.patch-dir".

```
(:patchable "reg: reg; patch;" "reg-sys")
```

See the section "Names of Patch Files", page 236.

:initial-status (**:initial-status** *status*) sets the initial status of the system when **make-system** is used to create the first major version. The system version-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.

Note: To change the status of the system recorded in the system version-directory file, call **make-system** with the **:update-directory** option. The **si:set-system-status** function also changes the status of a system but only as cached in memory.

:bug-reports (**:bug-reports** *list-name mouse-line-string*) specifies a name (a string) for mailing bug reports about the system. Zmail uses this name in its Bug Mail menu. Supply 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-Registrar.

```
(:bug-reports "Registrar"
              "Report problems with the Registrar system.")
```

:not-in-disk-label

:not-in-disk-label makes a patchable system not appear in the disk label comment. This option is useful for patchable systems internal to the main Lisp system, to avoid cluttering up the label.

:maintaining-sites

(**:maintaining-sites** *site-list*) specifies the list of sites that maintain the system. It declares which sites can patch a system and helps to monitor versions in order to ensure that no changes are lost. This option is meaningful only for patchable systems. For example:

```
(defsystem dla-file-system
  ...
  (:patchable)
  (:maintaining-sites :mit)
  ...)
```

The default for **:maintaining-sites** when it is undeclared is usually the local site. When you attempt to distribute a system with an undeclared maintaining site, you are warned and urged to supply a maintaining site. When you attempt to patch a system that is not maintained at your site, you will get a warning.

:component-systems

:component-systems specifies the names of other systems that are part of this system. Component systems cannot be patchable, that is, they must not specify the **:patchable** option in their **defsystem** declarations.

The component system will be made during the making of the containing system; however, the order of loading of the component system relative to the modules of the containing system is *not* guaranteed.

Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems; for example, assume that the **foo** system declaration declares a component system **bar** and that the user loads **foo** as follows: `(make-system 'foo :recompile)`. The recompiling operation will be performed on all the source files in both **foo** and **bar**. If **make-system** is called without options, then **foo** and **bar** are merely loaded. To compile the component system but not the containing system, compile the component system separately (for example, `(make-system "bar" :compile :noload)`) and then load the containing system: `(make-system 'foo)`.

The format of the **:component-systems** option is:
`(:component-systems` *name1 name2* `...)`

Example: The **:component-systems** specification in the **food-groups** declaration means that the **fruit** and **vegetables** systems will be made when **food-groups** is

made. If **food-groups** is compiled, then **fruit** and **vegetables** will also be compiled.

```
(defsystem fruit
  (:module citrus ("oranges" "grapefruit"))
  (:compile-load citrus)
  ...)

(defsystem vegetables
  (:module green ("broccoli" "lettuce"))
  (:compile-load green)
  ...)

(defsystem food-groups
  ...
  (:component-systems fruit vegetables)
  ...)
```

10.1 defsystem Modules

In addition to specifying the characteristics and transformations of the system, a **defsystem** declaration must also specify which files compose the system. For example, the **mysys** system contains only two files and has a **:compile-load** transformation (operation) performed on each file.

```
(defsystem mysys
  (:compile-load ("sys:george2;prog1" "sys:george2;prog2")))
```

It is customary, however, though not strictly required, to group a system's files into modules. A *module* is one or more files or modules that should be treated as a unit; transformations, like compilation or loading, operate on the module as a whole. The above declaration suffices only because the **:compile-load** transformation does not have any associated dependencies. Dependencies express relationships only between modules — not files.

Since the files of a module are treated as a unit, any one file in a module cannot depend on the previous compiling/loading of another file in that same module. If file "b" depends on file "a" having been loaded, then these files cannot be placed in the same module.

A common organizing principle for deciding which files should constitute a module is to group those files that perform a similar function, with the restriction that the files must not depend on one another. For example, all low-level definitions (variables and macros) might be placed in the same module.

The relationship of one **defsystem** module to another can be described as a hierarchy of dependencies. Modules often depend on other modules (in the same

system or in another system) having been previously loaded. The main program, for example, presumably depends on the previous loading of the low-level system definitions.

Once you correctly determine (1) which files should compose a module and (2) which modules depend on which other modules, you will never have to figure out these relationships again. **make-system** will compile and load your system correctly.

:module Option for defsystem

The **:module** option defines a module, which is a set of files or modules that should be treated as a unit. The included modules can be internal or external to the system being defined.

Example: The following code defines two modules. The definitions module consists of three files; the main module has only one file. When a module consists of more than one file, the file names must be specified as a list.

```
(:module definitions ("variables" "macros" "more-macros"))
(:module main "top-level")
```

The ordering of file or module names within a module is not significant. This rule also holds for the order of module declarations, with one exception: a module included in another module must be defined before it is used. The following example is correct because the mult module is declared before the main module refers to it. Main can refer to mult, but mult cannot refer to main.

```
(:module mult "mult") (:module main ("top" "comnds" mult "cometh"))
```

Along the same lines, all modules must be declared before any transformation can refer to them.

The format for using the **:module** option is:

(:module *name module-specification option...*)

name is the name of the module and is a symbol. Each module has one or more associated files, which can be referred to collectively by *name*. In other words, use *name* throughout the declaration instead of repeating the file names associated with the module.

module-specification specifies the files or modules named by the module and can be:

A string Represents a file name.

Example: The parser module consists of one file.

```
(:module parser "parser")
```

A symbol Represents a module name. The module name stands for all of the files that are in that module of the current system declaration.

The mult module is defined and then used within the specification of the main module.

```
(:module mult "mult")
(:module main ("top" "comnds" mult "cometh"))
```

An external module component

Expressed as a list of the form (*system-name module-names...*), to specify modules in another system. Each module name stands for all of its constituent files or modules. Example:

```
(defsystem fred
  ...
  (:module defs "defs")
  (:module foo (defs (zmail defs)))
  ...)
```

The **foo** module consists of two other modules: the **defs** module in the same system and the **defs** module in the **zmail** system. It is not generally useful to compile files that belong to other systems; thus, this **foo** module would not normally be the subject of a transformation. However, *dependencies* use modules and need to be able to refer to (depend on) modules of other systems. See the section "**defsystem** Transformations", page 201.

A list of module components

Can be a list of any valid module specification — a string, a symbol, or an external module component — or a list of file names. A list of module components is also known as an *anonymous module.*

Example: The user-interface module specification is a list of strings.

```
(:module user-interface ("ui-1" "ui-2"))
```

A list of file names is used when the names of the input and output files of a transformation are not related according to the standard naming conventions, for example, when a compiled code file has a different name (in more than just the file type) or resides in a different directory than the source file. The file names in the list are used from left to right, thus, the first name is the source file. Each file name after the first in the list is defaulted from the previous one in the list.

Example: The **prog** module consists of one file, but it lives in two directories, george and george2. Assuming that these are Lisp programs, that means that the file "q:>george>prog.lisp" is compiled into "q:>george2>prog.bin".

```
(:module prog (("q:>george>prog" "q:>george2>prog")))
```

Note the syntax of the example. To avoid ambiguity, a list of file names is allowed as a module component but *not* as a module specification.

Every file name is treated as if it were an infinite list of file names with the last file name, or, in the case of a single string, the *only* file name, repeated forever at the end.

Each simple transformation takes some number of input file name arguments, and some number of output file name arguments. As transformations are performed, these arguments are taken from the front of the file name list. The input arguments are actually removed, and the output arguments are left as input arguments to the next higher transformation. To make this clearer, consider having the **:compile-load** compound transformation performed on the **prog** module.

```
(:compile-load prog)
```

This means that **prog** is given as the input to the **:compile** transformation and the output from this transformation is given as the input to the **:fasload** transformation. The **:compile** transformation takes one input file name argument — the name of a Lisp source file — and one output file name argument — the name of the compiled code file. The **:fasload** transformation takes one input file name argument — the name of a compiled code file — and no output file name arguments. So, for the first and only file in the **prog** module, the file name argument list looks like ("q:>george>prog" "q:>george2>prog" "q:>george2>prog" ...). The **:compile** transformation is given arguments of "q:>george>prog" and "q:>george2>prog" and the file name argument list, which it outputs as the input to the **:fasload** transformation, is ("q:>george2>prog" "q:>george2>prog" ...). The **:fasload** transformation then is given its one argument of "q:>george2>prog".

The only valid **option** to the **:module** clause is **:package**. Sometimes a module needs to use the package specified by the attribute lists of its constituent files rather than the package declared for the system. The **:package** option to **:module** overrides the system package for the duration of the transformations performed on just this module.

Example: The mult module is compiled into the **tv** package, whereas the defs module is compiled to the **zwei** package.

```
(defsystem zmail
  ...
  (:package zwei)
  (:module defs "defs")
  (:module mult "mult" :package tv)
  ...)
```

10.2 defsystem Transformations

In addition to specifying the characteristics of and modules in the system, the **defsystem** form must declare all the transformations of the system. A **transformation** is an operation, such as compiling or loading, to be performed on a file or module *under certain circumstances.* These circumstances include (1) how the

user calls **make-system,** (2) the system dependencies; that is, how modules *depend* on the previous compiling/loading of other modules in the system, and (3) the specific *condition* that must be satisfied in order to trigger the transformation.

10.2.1 Interaction Between defsystem Transformations and make-system

The interaction between **make-system** and **defsystem** is quite complicated and resistant to generalizations; however, it is fair to state that **make-system** in many ways controls how and when a system is made, based on and restricted by the instructions in the system declaration. For example, **make-system** determines which transformations in the declaration take place; compile-type transformations do not occur unless the call to **make-system** explicitly specifies a compile-type option. The invocation `(make-system 'foo)` would load (by default) but not compile the system's files, regardless of whether or not the declaration specifies that the **:compile-load** transformation be performed on every module in the system. On the other hand, a request for recompilation (as in `(make-system 'foo :recompile)`) would not work in the unlikely case where the **defsystem** form declared only loading operations.

make-system also controls when each transformation takes place (that is, their order relative to one another) by looking at the system dependencies stated in the **defsystem** form; hence the order in which you declare transformations is not necessarily the order in which **make-system** performs them.

make-system must also determine whether the circumstances (*conditions*) are right for performing the requested and valid transformations in the system declaration. For example, a load operation will occur only if the test for loading is satisfied; this is usually "Is the binary file on disk newer than the version in the current world?" If not, then the load operation does not happen, regardless of the fact (1) that **make-system** by default loads files and (2) that the system declaration specifies that loading should occur. Every transformation has an associated condition.

The following oversimplified discussion of how **make-system** processes a **defsystem** form might clarify the above points. Consider the following brief extract from the declaration for system **fruit**; assume that the user only loads the system: `(make-system 'fruit)`. Also assume that the machine has been freshly booted, and so no files from the **fruit** system are in memory.

```
(defsystem fruit
  ...
  (:compile-load apple)
  (:fasload banana (:fasload apple))
  ...)
```

1. First **make-system** constructs a list of declared transformations: `(:compile-load apple)` breaks down into a compile of apple and a load of apple; each step is a separate transformation. `(:fasload banana)` is one transformation, a load of the banana module. So **make-system**'s list shows three transformations:

a. compile apple
b. load apple
c. load banana

Note: **make-system** performs only those transformations declared in **defsystem** form. Thus, it can never compile banana, since no compiling operation for banana is present.

2. The **make-system** function checks to see which options were requested by the user. In this case, none. **make-system** defaults to loading all files.

3. The next order of business is to match the requested option (by default, a load) against the current list of transformations to see which of them should be performed in this execution of **make-system**. In this example, it matches load against the three transformations, revising the list like so:

   a. load apple
   b. load banana

4. Next **make-system** examines the system dependencies one at a time. Does the loading of apple depend on the previous loading of another module? No. It asks the same question about the loading of banana. This time the answer is yes. The load of banana depends on `(:fasload apple)`. **make-system** checks to see that the transformation is going to be done. If not, it adds the transformation (required by this dependency) to its proper place in the to-do list. In this example, the dependency requires apple to be loaded before banana, thus the current list is correct as it stands.

   a. load apple
   b. load banana

   A more complicated declaration with more dependencies might require **make-system** to shuffle the list to account for the dependencies. In this simple example, the order of declaration corresponds to the order of loading.

5. Finally, **make-system** looks at the condition of each transformation. If the condition (test) is met, **make-system** performs the transformation; otherwise, it deletes the transformation from its list. In the example, the user has not stated any conditions, so **make-system** uses the predefined conditions. For loading, the default condition is "load the binary file on disk if, and only if, it is newer than the version in the current world".

   make-system looks at the first transformation, a load of apple. Since no files from **fruit** are in memory, the condition is satisfied, and apple is loaded.

   make-system looks at the second transformation, a load of banana. Again the condition obtains; banana is loaded.

Types of defsystem Transformations

Transformations fall into two categories: simple and compound. A *simple transformation* is a single operation on a file. **:fasload** is an example of a simple transformation and means load the specified binary files of the indicated module. A *compound transformation* takes the output from one transformation and performs another transformation on it. **:compile-load** is such a compound transformation and means load the indicated module, compiling it first under certain conditions.

Example 1: The **:fasload** simple transformation is performed on the foo module; the binary version of the files composing the foo module are loaded.

```
(:module foo "foo")
(:fasload foo)
```

Example 2: The **:fasload** simple transformation is performed on the output produced from the **:compile** transformation being performed on the foo module.

```
(:module bar ("bar-1" "bar-2"))
(:module foo ("foo-1" bar "foo-3"))
(:fasload (:compile foo))
```

Example 3: The **:compile-load** compound transformation is performed on the files "foo" and "bar". First the files are compiled; the output is loaded.

```
(:compile-load ("foo" "bar"))
```

Note that in specifying a transformation the user must supply the *name* of the transformation and the *input* to the transformation, in that order.

- The name is one of the predefined transformations, like **:fasload**, **:compile**, or **:compile-load**, or a user-defined transformation.
- The input is usually a module name, a module specification, or another transformation whose output is used.

Examples:

```
(:fasload foo)

(:fasload ("foo" "bar"))

(:fasload (:compile foo))
```

Dependencies and Conditions

In addition to name and input, which are required, a transformation specification can optionally include dependencies and a condition. A *dependency* declares that under specified circumstances all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. A *condition* is a predicate that states in what circumstance the transformation should occur. Each transformation has a default condition; hence, most users do not explicitly include the condition of a transformation.

No general rules supply the exact format for every transformation, although there are common conventions; the arguments are defined individually by the Lisp form that defines the transformation. For a simple transformation, the general format is:

(*name input dependencies condition*)

Example: The elements of the **:fasload** transformation are labelled.

```
(:compile-load module-1)
(:fasload module-2 (:compile module-1) 'file-exists-p)

  name     input    dependency          condition
   |         |         |                    |
   |         |         |                    |
   ↓         ↓         ↓                    ↓
  (:fasload module-2 (:compile module-1) 'file-exists-p)
```

:fasload is the name of the transformation performed on module-2, the input. **(:compile module-1)** is a load dependency of the **:fasload** transformation and means loading module-2 depends on module-1 having been compiled. The previous transformation `(:compile-load module-1)` explicitly declares this compile operation. The condition asks the question "Does the file on which the transformation is applied exist?" If the condition is satisfied, the transformation proceeds.

Note two important points:

- Transformations can refer only to other transformations that have been previously declared. The transformation on module-2 can refer to (depend on) the transformation on module-1, but not vice versa.

- The format for expressing a dependency is itself a transformation specification, which is either a list pairing a transformation and a module name (for example, `(:compile module-1)`) or a list of such lists `((:compile module-1) (:compile module-2))`. The module name is either a symbol that is the name of a module in the current system or an external module component.

The format of the most commonly used compound transformation, **:compile-load**, looks like:

(*name input compile-dependencies load-dependencies compile-condition load-condition*)

Consider the following **defsystem** form, particularly the **:compile-load** transformation specification marked with an asterisk. Example:

```
(defsystem registrar
  ...
  (:module main "top-level")
  (:module definitions ("variables" "macros" "more-macros"))
  (:module resources "resources")
```

```
  (:compile-load definitions)
* (:compile-load resources (:fasload definitions))
  (:compile-load main (:fasload definitions resources)))
```

In this case, the **:fasload** transformation on the definitions module is a dependency of the **:compile-load** transformation on the resources module. To be more precise, the **:fasload** transformation functions as a *compile-time* dependency, meaning that it applies only when the call to **make-system** includes a compile-type option, as in **(make-system 'registrar :compile)**. The example does not explicitly state a condition, so the default condition for the **:fasload** transformation — **si:file-newer-than-file-installed-p** — is assumed. This condition is true when the file version on disk is newer than the version in the current world. If the condition is satisfied the transformation is performed.

Transformation specifications can also include *load-time* dependencies.

Example: Suppose you define a system to create and access a number of data structures. The structures (defined by **defstruct**) for accessing these data structures live in the structure-defs module; the code for creating the structures lives in the module structure-making-functions.

The structures module needs both (1) the **defstructs** declared in structure-defs to compile the code that accesses the data structures, and (2) the special constructor functions in structure-making-functions to create the data structures when it (the structures module) is loaded by **make-system**.

Given these dependencies, consider the following transformations.

```
(:compile-load structure-defs)
(:fasload structure-making-functions)
(:compile-load structures (:fasload structure-defs)
                          (:fasload structure-making-functions))
```

In the third transformation `(:fasload structure-defs)` is the compile-time dependency of the compiling phase of **make-system**. `(:fasload` structure-making-functions) is the load-time dependency of the loading phase of **make-system**.

Assuming that **make-system** is called with the **:compile** option, what is the effect of these compile and load dependencies on the actual making of the system? The result is that the following operations, *if they occur*, will be performed in this relative order. During the compile phase of the **make-system**:

- Load the structure-defs module.
- Compile the structures module.

During the load phase of the **make-system**:

- Load the structure-making-functions module.
- Load the structures module.

The phrase *if they occur* in the preceding paragraph refers to the effect of conditions on whether a transformation occurs. The default condition for compiling is "compile the source file if it is newer than its binary version". For loading, the equivalent default is "load the binary file on disk if it is newer than the version in the current world". Hence the items above can be more precisely written, as follows:

- Load the structure-defs module unless the newest version is already loaded.
- Compile the structures module unless the newest version is already compiled.

During the load phase of the **make-system**:

- Load the structure-making-functions module unless the newest version is already loaded.
- Load the structures module unless the newest version is already loaded.

The dependency must be a transformation that was *explicitly* specified as a transformation in the system definition, not just an action that might have been performed incidentally. That is, if you have a dependency **(:fasload foo)**, it means that **(:fasload foo)** must be explicitly declared as a transformation of your system; it does not simply mean that you depend on **foo** being loaded. Strictly speaking, you do not declare `(:fasload foo)`. It is sufficient if a compound transformation, such as **:compile-load**, expands into the required transformation on the specified module, such as in the third example below.

It is not sufficient, however, if the action is performed as part of a transformation on an anonymous module constructed of other modules, such as in the second example below. However, the following is correct and works properly:

```
(defsystem foo
  (:module foo "foo")
  (:module bar "bar")
  (:compile-load (foo bar)))
```

But the following example will signal an error because **foo**'s **:fasload** does not occur. The loading of **foo** is performed only implicitly as part of the **:fasload** transformation on the anonymous module **(foo bar)** implicit in the **(:compile-load (foo bar))**.

Wrong:
```
(defsystem foo
         (:module foo "foo")
         (:module bar "bar")
         (:module blort "blort")
         (:compile-load (foo bar))
         (:compile-load blort (:fasload foo)))
```

You must instead write:

Right:
```
(defsystem foo
  (:module foo "foo")
  (:module bar "bar")
  (:module blort "blort")
  (:compile-load foo)
  (:compile-load bar)
  (:compile-load blort (:fasload foo)))
```
In the above example, **(:fasload foo)** is part of the expansion of **(:compile-load foo)**; therefore, it can be used as a dependency.

Dependencies are neither transitive nor inherited. For example, suppose module one depends on macros defined in module two, and therefore needs two to be loaded in order to compile. Suppose also that module two has a similar dependency on module three. Module three does not loaded automatically during the compilation of one, because the system facility does not assume that module one also depends on module three. Transformations with these dependencies would be written as follows:
```
(:fasload three)
(:compile-load two (:fasload three))
(:compile-load one (:fasload two))
```

To express the relationship that the compilation of module one depends on both two and three, supply the whole history of dependencies:
```
(:fasload three)
(:compile-load two (:fasload three))
(:compile-load one (:fasload two three))
```

If, in addition, one depended on three during loading, but not on two (perhaps one contains **defvars** whose initial values depend on functions or special variables defined in module three), the transformations would be written as follows:
```
(:fasload three)
(:compile-load two (:fasload three))
(:compile-load one (:fasload two three) (:fasload three))
```

So far nothing has been said about what can be given as a *condition* for a transformation, except for the default functions that check for a source file being newer than the binary, and so on. In general, any function that takes the same arguments as the transformation function (for example, **compile-file**) and returns **t** if the transformation needs to be performed, can be in this place as a symbol, including, for example, a closure.

For example, suppose the user defines a function called **file-exists-p**, whose purpose is to determine whether its argument, a file name, exists in the file system. If the file exists, the function returns **t**; otherwise it returns **nil**. In this case, the transformation would be written as follows:
```
(:module another-file "another-file")
(:module the-file "the-file")
(:fasload the-file (:fasload "another-file") 'file-exists-p)
```

To specify the condition without the dependency, write the transformation with this modification:

```
(:fasload the-file () 'file-exists-p)
```

To take another example, suppose a file contains **compile-flavor-methods** for a system and should therefore be recompiled if any of the flavor method definitions change. In this case, the condition function for compiling that file should return **t** if either the source of that file itself or any of the files that define the flavors have changed. This is the purpose of the **:compile-load-init** compound transformation, which is defined in the **si** package like this:

```
(defmacro (:compile-load-init defsystem-macro)
          (input add-dep &optional com-dep load-dep
           &aux function)
  (setq function (let-closed ((*additional-dependent-modules*
                                (parse-module-components
                                  add-dep
                                  *system-being-defined*)))
                   'compile-load-init-condition))
  `(:fasload (:compile ,input ,com-dep ,function) ,load-dep))


(defun compile-load-init-condition (source-file binary-file)
  (or (file-newer-than-file-p source-file binary-file)
      (local-declare ((special *additional-dependent-modules*))
        (other-files-newer-than-file-p
          *additional-dependent-modules*
          binary-file))))
```

The condition function generated when this macro is used returns **t** either if **file-newer-than-file-p** would do so with those arguments, or if any of the other files in **add-dep** (which presumably is a *module specification*) are newer than the compiled code file. Thus the file (or module) to which the **:compile-load-init** transformation applies will be compiled if it or any of the source files on which it depends has been changed, and will be loaded under the normal conditions. In most (but not all cases), **com-dep** would be a **:fasload** transformation of the same files as **add-dep** specifies, so that all the files on which this one depends would be loaded before compiling it.

10.2.2 List of defsystem Transformations

:fasload **:fasload** calls the **si:load-binary-file** function to load the indicated files, which must be compiled code files. The condition defaults to **si:file-newer-than-installed-p**, which is **t** if a newer version of the file exists on the file computer than was read into the current environment.

:readfile **:readfile** calls the **readfile** function to read the indicated files. Use this for loading files that are not to be compiled, that is, interpreted code. The condition defaults to **si:file-newer-than-installed-p**.

:compile **:compile** calls the **compiler:compile-file** function to compile the

indicated files. The condition defaults to **si:file-newer-than-file-p,** which returns **t** if the source file has been written more recently than the compiled code file.

:load-bfd **:load-bfd** explicitly loads a font that is not in one of the system font directories by calling the **fed:read-font-from-bfd-file** function to load the specified font file(s). See the section "Font Basic Concepts" in *Text Editing and Processing*. The condition defaults to **bfd-file-newer-than-installed-p,** which returns **t** if a version of the font file exists that is newer than the installed version.

:do-components

(:do-components *dependencies*) inside a system with component systems causes the *dependencies* to be done before anything in the component systems. This is useful when you have a module of macro files used by all of the component systems. Example:

```
(defsystem example
  ...
  (:component-system foo)
  (:module macros "macros")
  (:compile-load macros)
  (:do-components (:fasload macros))
  ...)
```

:compile-load

:compile-load is the most commonly used compound transformation.
/(**:compile-load** *input compile-dependencies load-dependencies compile-condition load-condition*)
is the same as
/(**:fasload** (**:compile** *input compile-dependencies compile-condition*) *load-dependencies load-condition*)

All arguments after *input* are optional. The compile condition determines whether or not the compile dependency will be performed; the load condition does the same for the load dependency.

Example: Suppose you define a system to create and access a number of data structures. The structures (defined by **defstruct**) for accessing these data structures live in the structure-defs module; the code for creating the structures lives in the module structure-making-functions.

The structures module needs both (1) the **defstructs** declared in structure-defs to compile the code that accesses the data structures, and (2) the special constructor functions in structure-making-functions to create the data structures when it (the structures module) is loaded by **make-system**.

These dependencies of the structures module are expressed in the following **:compile-load** transformation.

```
(:compile-load structure-defs)
(:fasload structure-making-functions)
(:compile-load structures (:fasload structure-defs)
                          (:fasload structure-making-functions))
```

`(:fasload structure-defs)` is the compile-time dependency of the compiling phase of **make-system**. `(:fasload` structure-making-functions) is the load-time dependency of the loading phase of **make-system**.

Assuming that **make-system** is called with the **:compile** option, what is the effect of these compile and load dependencies on the actual making of the system? The result is that the following operations, *if they occur*, will be performed in this relative order. During the compile phase of the **make-system**:

- Load the structure-defs module.
- Compile the structures module.

During the load phase of the **make-system**:

- Load the structure-making-functions module.
- Load the structures module.

The compile and load conditions determine whether or not the transformations, compiling and loading, occur. The default condition for compiling is "compile the source file if it is newer than its binary version". For loading, the equivalent default is "load the binary file on disk if it is newer than the version in the current world." Hence, to be more precise, the load transformations above will be performed unless the newest binary version is already loaded; the compile transformations will be performed unless the newest version is already compiled.

:compile-load will not suffice in certain situations. For example, assume one of the structure definitions in structure-defs changes. In order for this change to be reflected in the compiled code for structures module, the structures module must be recompiled. The transformation in the example does not in any way guarantee that recompilation will occur. In this situation use **:compile-load-init**.

:compile-load-init

:compile-load-init compiles the file (or module) to which this transformation is applied if the file (or module) or any of the source files on which it depends has been changed, and will load it under normal conditions. Its format is:

(:compile-load-init *input modules-to-test-for-file-newer-than-file-p compile-dependency load-dependency*)

modules-to-test-for-file-newer-than-file-p must be specified as a list. The most common use of **:compile-load-init** is to recompile files containing **compile-flavor-methods** for a system when the definitions of the flavor methods change.

Example 1: cometh is a module that compiles flavor methods. (streams defs envr builder devices) is the list of modules to be tested to determine whether any source file is newer than its associated binary file; these modules contain all the flavor definitions and methods for the defined system. The effect of **:compile-load-init** is to recompile cometh whenever flavor definitions or methods have changed. The **:fasload** transformation is a compile dependency, ensuring that definitions and methods are loaded when compiling the compile flavor methods.

```
(:compile-load-init cometh
(streams defs envr builder devices)  ; test dependency
(:fasload streams defs envr builder devices))) ; compile dependency
```

Consider another example, which illustrates why **:compile-load-init** is preferred to **:compile-load** in some situations.

Example 2: Suppose you define a system to create and access a number of data structures. The structures (defined by **defstruct**) for accessing these data structures live in the structure-defs module; the code for creating the structures lives in the module structure-making-functions.

The structures module needs both (1) the **defstruct**s declared in structure-defs to compile the code that accesses the data structures, and (2) the special constructor functions in structure-making-functions to create the data structures when it (the structures module) is loaded by **make-system**.

These dependencies of the structures module are expressed in the following **:compile-load** transformation.

```
(:compile-load structure-defs)
(:fasload structure-making-functions)
(:compile-load structures (:fasload structure-defs)
                          (:fasload structure-making-functions))
```

However, a problem arises when, for example, one of the structure definitions in structure-defs changes. In order for this change to be reflected in the compiled code for structures module, the structures module must be recompiled. The transformation in the example does not in any way guarantee that recompilation will occur.

The solution is to use **:compile-load-init** to add a test dependency on structure-defs, to guarantee that structures is compiled if structure-defs has changed.

```
(:compile-load-init structures (structure-defs)
 (:fasload structure-defs)
 (:fasload structure-making-functions))
```

The list (structure-defs) indicates the dependency "when structure-defs changes".

10.2.3 :skip defsystem Macro

It is sometimes useful to specify a transformation upon which something else can depend, which is not performed by default, but rather only when requested because of that dependency. The transformation nevertheless occupies a specific place in the hierarchy.

The **:skip defsystem** macro allows specifying a transformation of this type. For example, suppose a special compiler for the readtable is not ordinarily loaded into the system; the compiled version should still be kept up to date, and it needs to be loaded if the readtable ever needs to be recompiled.

```
(defsystem reader
  (:pathname-default "AI: LMIO;")
  (:package system-internals)
  (:module defs "RDDEFS")
  (:module reader "READ")
  (:module read-table-compiler "RTC")
  (:module read-table "RDTBL")
  (:compile-load defs)
  (:compile-load reader (:fasload defs))
  (:skip :fasload (:compile read-table-compiler))
  (:rtc-compile-load read-table (:fasload read-table-compiler)))
```

Assume that there is a compound transformation **:rtc-compile-load** that is like **:compile-load**, except that it is built on a transformation called something like **:rtc-compile**, which uses the readtable compiler rather than the Lisp compiler. In the above system, then, if the **:rtc-compile** transformation is to be performed, the **:fasload** transformation must be done on **read-table-compiler** first; that is, the readtable compiler must be loaded if the readtable is to be recompiled. If you say **(make-system 'reader ':compile)**, then the **:compile** transformation will still happen on the **read-table-compiler** module, compiling the readtable compiler if necessary. But if you issue **(make-system 'reader)**, the reader and the readtable will be loaded, but the **:skip** keeps this from happening to the readtable compiler.

10.3 Adding New Options to defsystem

Options to **defsystem** are defined as macros on the **si:defsystem-macro** property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return **nil**. They can use several variables, but the only one of general interest is **si:*system-being-defined***.

si:*system-being-defined* *Variable*

The internal data structure representing the system that is currently being constructed.

si:define-defsystem-special-variable *name form* *Special Form*

Causes *form* to be evaluated and *name* to be bound to the result during the expansion of the **defsystem** special form. This allows you to define new variables similar to **si:*system-being-defined***.

si:define-simple-transformation *name function default-condition input-file-types output-file-types* &optional *pretty-names* (*compile-like* **t**) (*load-like* **nil** *ll-p*) *Special Form*

This is the most convenient way to define a new simple transformation. For example,

```
(si:define-simple-transformation :compile si:compile-file-1
        si:file-newer-than-file-p
        (:lisp) (:bin))
```

input-file-types and *output-file-types* arguments for a transformation specify how many input file names and output file names to receive as arguments (in this example one of each).

pretty-names, an optional argument, specifies how the transformation will be printed in messages to the user. It can be a list of the imperative ("Compile"), the present participle ("Compiling"), and the past participle ("compiled"). Note that the past participle is not capitalized, because it is not used at the beginning of a sentence. *pretty-names* can be just a string, which is taken to be the imperative, and the system will conjugate the participles itself. If *pretty-names* is omitted or **nil** it defaults to the name of the transformation.

compile-like and *load-like*, both optional arguments, specify when the transformation should be performed. Compile-like transformations are performed when the **:compile** keyword is given to **make-system**. Load-like transformations are performed unless the **:noload** keyword is given to **make-system**. By default *compile-like* is **t** but *load-like* is **nil**. If you do not specify *load-like*, it defaults to the boolean inverse of the *compile-like* argument.

Complex transformations are just defined as normal macro expansions, for example,

```
(defmacro (:compile-load si:defsystem-macro)
                (input &optional com-dep load-dep
                                 com-cond load-cond)
  `(:fasload (:compile ,input ,com-dep ,com-cond)
             ,load-dep ,load-cond))
```

11. Loading the System Definition

11.1 Loading System Definitions That Use Logical Pathnames

Once you have written a large program and defined it as a system, you want **make-system** (or the relevant Command Processor commands that call **make-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 **make-system** to be able to find and load your system:

- System file, named sys:site;*system-name*.system file
- Translations file, named sys:site;*logical-host*.translations file
- 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.

11.1.1 Sys:site;*System-name*.System File

make-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 `(make-system 'graphic-lisp)` it looks for the file sys:site;graphic-lisp.system.

The system file contains two forms and looks like this:

(**fs:make-logical-pathname-host** *"logical-host"*)
(**si:set-system-source file** *"system-name"*
"logical-host:logical-directory; system-name")

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")
(si: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, (**si: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 **si:set-system-source-file**, which specifies the logical pathname of the system declaration file. **make-system**, after referring to the translation definitions, loads the system declaration file.

11.1.2 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: FILE-SYSTEM -*-

(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**.

11.1.3 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.

A sample system declaration file might look like the following:

```
;;; -*- Mode: LISP; Package: USER; -*-
;;; Created 3/08/82 05:09:13 by PGBRUCE

(defpackage registrar global 2000.))

;;; definition of the registrar system

(defsystem registrar
  (:name "Automatic Registration System")
  (:pathname-default "reg:reg;")
  (:not-in-disk-label)

  (:module main "top-level")
  (:module definitions ("variables" "macros" "more-macros"))
  (:module resources "resources")

  ;; transformation should be performed on each module.
  (:compile-load definitions)
  (:compile-load resources (:fasload definitions))
  (:compile-load main (:fasload definitions resources)))
```

Note the attribute list. The system declaration file is always a lisp-mode file and is compiled into the **user** package.

The name of the system declaration file does not require an exact format, since you explicitly specify the pathname in the **si: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 **make-system** the **si:set-system-source-file** form loads the system declaration file, specifically the .newest version.

11.2 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 **si:set-system-source-file** form. Have your init file evaluate the form (or type the form at a Lisp Listener) prior to calling **make-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.

12. Making a System

Making a system means compiling or loading that system according to the specifications in the **defsystem** form and to the user options supplied in the making of the system. The relationship of **make-system** to the **defsystem** form which it processes is quite complex. Please read the following section before making your system: See the section "Interaction Between **defsystem** Transformations and **make-system**", page 202.

The tools provided for making a system are the **make-system** function or the Command Processor commands Load System or Compile System.

make-system *name* &rest *keywords* *Function*

make-system reads the system declaration file for system *name* and then performs the transformations it specifies as well as any other instructions you supply in the call to **make-system**. A call to **make-system** without any options loads the existing binary files for the system *name*, as in

```
(make-system 'mysys)
```

If **mysys** is a patchable system, **make-system** loads the binary files of the released version of the system, if one exists; otherwise it loads the binary files of the latest version. **make-system** examines the system version-directory file to determine which files are in which version of the system.

Supply one or more optional keyword arguments to alter the default behavior of **make-system**. For example, to compile the source files that are newer than their corresponding binary files, and increment the major version, type:

```
(make-system 'mysys :compile)
```

If **mysys** is a patchable system **make-system** also updates the system version-directory file, making the just-compiled version the latest version of the system.

By default **make-system** displays a message listing what transformations it is going to perform on what files. It asks you for confirmation and then performs the transformations. Prior to each transformation a message is printed listing the transformation being performed, the file to which it is being done, and the package. For example:

```
Load all twenty-six of them? (Y, N, or S)
```

If you answer S (meaning *selective*), you are asked for confirmation of each individual transformation. Note: you can suppress these messages by supplying the **:noconfirm** option in the **make-system**. This keyword assumes an affirmative answer to all questions.

```
(make-system 'mysys :compile :noconfirm)
```

If you run **make-system** on a system that is patchable and not already loaded, **make-system** calls **load-patches** after loading the system. **load-patches** is called with the same options as **make-system**; for example, if **make-system** is specified with the **:silent** keyword, **load-patches** is also silent.

make-system supports a number of *keyword* options, which modify its behavior.

make-system Keywords

The **make-system** function recognizes many keywords, which can be characterized as query keywords, operation keywords, keywords for patchable systems, and miscellaneous keywords.

The query keywords ask you questions or suppress the asking of questions during **make-system**. By default **make-system** prompts you about each operation it intends to perform on a file and then reports each operation as it does it.

:batch — **:batch** allows a large compilation to be done unattended. It acts like **:noconfirm** with regard to questions, turns off MORE processing and sets **inhibit-fdefine-warnings** to **t**, and saves the compiler warnings in an editor buffer and a file (it asks you for the name).

:noconfirm — **:noconfirm** suppresses all questions that you would otherwise be asked, assuming an affirmative answer for each question.

:nowarn — **:nowarn** suppresses questions requiring operator response. Otherwise you must give permission to have straightforward tasks (like reading files) performed.

:selective — **:selective** asks the user whether or not to perform each transformation that appears to be needed for each file.

:silent — **:silent** suppresses reporting of each transformation as it occurs.

The operation keywords determine what operations **make-system** will perform on the files in the system, subject to the constraints in your system definition. By default **make-system** just loads the latest files.

:compile — **:compile** compiles the newest versions of the source files if and only if they are newer than the compiled code files. It also loads the compiled code files. It increments the system major version and sets the minor version to zero. For patchable systems **:compile** updates the system version-

directory file, making the just-compiled version the latest version of the system.

:noload — **:noload** does not load any files except those required by dependencies. **:noload** is used in conjunction with **:compile**, so that you can compile files but not load them unless it is necessary for compiling a subsequent file in the system definition.

:recompile — **:recompile** compiles and loads all files, regardless of whether or not they need to be compiled or loaded. It increments the system major version and sets the minor version to zero. For patchable systems **:recompile** updates the system version-directory file, making the just-compiled version the latest version of the system. **:recompile** has the effect of **:compile** and **:reload**. **:recompile** always compiles the newest versions of each file.

:reload — **:reload** bypasses the specified conditions for performing a transformation. Thus files are loaded even if they are not newer than the installed version.

These miscellaneous keywords are not used frequently but are included here for completeness.

:noop — **:noop** is ignored. This is mainly useful for programs that call **make-system**, so that such programs can include forms like:

```
(make-system 'mysys (if compile-p ':compile ':noop))
```

:print-only — **:print-only** displays the transformations that would be performed; does not actually do any compiling or loading.

The following keywords are valid only for patchable systems, that is, those defined with the **:patchable** option in the system definition.

:increment-patch
:increment-patch increments a patchable system's major version without doing any compilation.

:no-increment-patch
When given along with the **:compile** option, **:no-increment-patch** disables the automatic incrementing of the major system version that would otherwise take place. Note that the **:no-increment-patch** keyword must follow **:compile** in the **make-system** declaration.

:version — **:version** loads specific versions of a patchable system. Versions are described in the system version-directory file as a number, a name, and/or a keyword — newest, released, or latest.

:version takes an argument, which you must supply as a list. For example, to load version 34 of **mysys**, invoke:

```
(make-system 'mysys '(:version 34.))
```

Argument	*Meaning*

:released	Loads the major and minor system designated by the system maintainer as the released version. See the section "Types of Patch Files", page 233.
	
	When you do not supply the **:version**, **:compile**, or **:recompile** keyword, **make-system** always loads the released system. If there is no released version, then **make-system** loads the latest version. It also loads patches for the version of the system you specify.
	
	Example: To load the released version of **george**, type:
	`(make-system 'george '(:version :released))`
	or the simpler form:
	`(make-system 'george)`
	
	Note: The system maintainer designates a particular version of the system as the released version by using the **:update-directory** keyword to **make-system**.

:latest	Loads the system designated by the system developer as the latest version. This version has the highest major and minor version numbers. The most recently compiled version of the system is automatically assigned the designation **:latest** in the system version-directory file. Example: On Monday the system developer compiles the most up-to-date source files in the **mailer** system and then loads each newly compiled file, as follows: `(make-system 'mailer ':recompile)` **make-system** also automatically updates the system version-directory file, marking Monday's version of **mailer** as the latest version.

On Tuesday the system developer wants to load the version that was compiled the day before; hence:

```
(make-system 'mailer '(:version :latest))
```

System developers typically use the **:latest** keyword to load systems under development.

:newest Loads the most recently compiled version of each *file* of a system. The distinction between newest and latest is subtle; the newest version differs from the latest version when individual files in the system have been compiled by hand. Note that you cannot make or load patches for the newest system.

Example: On Tuesday the system developer loads the latest version of the system **alphabet**, which contains files A.lisp.10, A.bin.10, B.lisp.10, B.bin.10, and so on, to Z.lisp.10, Z.bin.10. The developer makes changes to several functions in A.lisp.10, compiles the file to A.bin.11, and saves the source file, A.lisp.11.

On Wednesday the developer wants to test the incremental changes to the system, but, to be cautious, doesn't want to destroy the latest system that was compiled and loaded on Monday. To do so, the developer uses the **:newest** keyword to load a system consisting of the most recently compiled versions of each of the system's files: A.bin.11 and the remaining files, B.bin.10 through Z.bin.10.

```
(make-system 'alphabet '(:version :newest))
```

The latest version remains intact; and the newest version is the most experimental version of the system.

version-number Loads a particular major version number of the system.

```
(make-system 'george '(:version 23.))
```

Note the decimal point after the version number.

version-name Loads the particular version of the system known as *:version-name* in the system version-directory file. This name is a user-defined symbol. The system maintainer must have previously assigned the version name by using the **:update-directory** keyword to **make-system**.

Example: The system developer plans to demonstrate the **frog** system to a group of prospective customers from Japan. Aside from the regular debugged version, there is a special version that works in Japanese.

After assigning the version name **:japanese** to this particular version of **frog**, the developer can load it, as follows:

```
(make-system 'frog '(:version :japanese))
```

:update-directory

:update-directory updates the system version-directory file for the currently loaded version of the system. Use **:update-directory** to designate the currently loaded version as the released version or to give it a name of your choice. Once you have updated the system version-directory file, you can use the **:version** option for **make-system** to load that system.

:update-directory takes a keyword argument (like **:released**) and assigns that keyword to the currently loaded version of the system. Specified without an argument, **:update-directory** enters the **:latest** keyword in the system version-directory file. The argument, if any, must be supplied as a list.

Example 1: The system developer wants to release the latest version of **mail**, version #34, for general use. There is currently no released version. The following form loads the latest version of **mail** and designates it as the released version.

```
(make-system 'mail '(:update-directory :released))
```

The developer could also have given this longer but equivalent form:

```
(make-system 'mail '(:version 34.) '(:update-directory :released))
```

Example 2: The system developer plans to demonstrate the

frog system to a group of prospective customers from Japan. Aside from the regular debugged version, there is a special version that works in Japanese. The developer decides to assign this special version a *version-name* of **:japanese**. The system is already loaded, so the developer invokes:

```
(make-system 'frog ':noload '(:update-directory :japanese))
```

To load this version in the future the developer must use the *version-name* argument to the **:version** keyword, like so: `(make-system 'frog '(:version :japanese)`.

Example 3: The system maintainer wants to maintain a version of the **graphic-lisp** system across incompatible releases, such as Symbolics Release 4.5 and 5.0.

```
(make-system 'graphic-lisp '(:update-directory :rel-4-5))
```

Users can now run **graphic-lisp** under the older release by invoking **make-system** with the version name of **:rel-4-5**.

12.1 Adding New Keywords to make-system

A user-defined keyword is a mechanism for communicating with a transformation. Defining new keywords sets up variables that **make-system** transformations look at during their execution. The effect is to alter the way the transformation works.

For example, during the loading of the system files the function **fdefine** is called. **fdefine** in turn looks at the value the variable **inhibit-fdefine-warnings** to determine whether it should warn the user when a function spec is begin redefined by a file different from the one that defined it originally. You decide to add a new **make-system** keyword called **:just-warn** whose intent is to display the **fdefine** warnings for functions being overwritten but not to query the user, as **fdefine** normally does when the value of **inhibit-fdefine-warnings** is **nil**.

The first step is use the special form **si:define-make-system-special-variable** to define a new variable. Because the variable is bound within the call to **make-system** you can define new variables that are very similar to those already in existence.

si:define-make-system-special-variable *name form* &optional (*defvar-p* **t**) *Special Form*

Causes the variable *name* to be bound to *form*, which is evaluated at **make-system** time, during the body of the call to **make-system**. This allows you to define new variables similar to those that already exist. If you specify *defvar-p* as (or defaulted to) **t**, *name* is defined with **defvar**. It is not given an initial value. If *defvar-p* is specified as *nil*, *name* belongs to some other program and is not **defvar**ed here.

In the following example, **inhibit-fdefine-warnings** is bound to itself within the call to **make-system**.

```
(si:define-make-system-special-variable
   inhibit-fdefine-warnings inhibit-fdefine-warnings nil)
```

make-system keywords are defined as functions on the **si:make-system-keyword** property of the keyword. The functions have no arguments. The following example sets the value of **inhibit-fdefine-warnings** to the new keyword **:just-warn**.

```
(defun (:just-warn si:make-system-keyword) ()
  (setq inhibit-fdefine-warnings ':just-warn
        batch-mode-p* t
        *query-type* ':noconfirm))
```

make-system keywords can have effect either directly when called or by pushing a form to be evaluated onto the list **si:*make-system-forms-to-be-evaled-after***, **si:*make-system-forms-to-be-evaled-before***, or **si:*make-system-forms-to-be-evaled-finally***. However, in general, the only useful thing to do is to set some special variable defined by **si:define-make-system-special-variable**.

User-defined transformations can also have their behavior controlled by new special variables, which can be set by new keywords. For example, if you want to get at the list of transformations to be performed, the right way would be to set **si:*file-transformation-function*** to a new function, which then might call **si:do-file-transformations** with a possibly modified list. That is how the **:print-only** keyword works.

Remember that when you execute **make-system**, it adds the loaded system to the system version-directory file of patchable systems unless you specify certain keywords that explicitly suppress this action. For example, **:print-only** is among these keywords. Certain user-defined keywords — those that rebind **si:*file-transformation-function*** and then recursively call **make-system** — must also take into account this updating feature of **make-system**. The following code is assumed to be in the **si** package.

```
(defun (:print-only make-system-keyword) ()
  (no-update-directory)                        ;Suppresses updating
  (setq *file-transformation-function* 'print-file-transformations))
```

Some of the variables bound by the predefined **make-system** keywords are:

si:*system-being-made* *Variable*

The internal data structure that represents the system being made.

si:*make-system-forms-to-be-evaled-before* *Variable*

A list of forms that are evaluated before the transformations are performed.

si:*make-system-forms-to-be-evaled-after* *Variable*

A list of forms that are evaluated after the transformations have been performed.

si:*make-system-forms-to-be-evaled-finally* *Variable*

A list of forms that are evaluated after the body of **make-system** has completed. This differs from **si:*make-system-forms-to-be-evaled-after*** in that these forms are evaluated outside of the "compiler context", which sometimes makes a difference.

si:*query-type* *Variable*

Controls how questions are asked. Its normal value is **:normal**. **:noconfirm** means no questions are asked, and **:selective** asks a question for each individual file transformation.

si:*silent-p* *Variable*

If **t**, no messages are displayed.

si:*batch-mode-p* *Variable*

If **t**, **:batch** was specified.

si:*redo-all* *Variable*

If **t**, all transformations are performed, regardless of the condition functions.

si:*top-level-transformations* *Variable*

A list of the names of transformations that will be performed, such as **(:fasload :readfile)**.

si:*file-transformation-function* *Variable*

The actual function that gets called with the list of transformations that need to be performed. The default is **si:do-file-transformations**.

13. Patch Facility

Software development is a process of *incremental* changes to *many* large programs by *many* developers followed by the *uniform* distribution of these changes 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 Lisp Machine editor, provides special tools that make this process relatively easy and disaster-free for the developer.

The important point is that it is the patch files — and *not* the newly written source files — that allow the changes to be put into widespread use immediately. The reason for this rests with the size of most software systems and the way they are distributed. Because loading all the files in a large system is so time-consuming, the system maintainer might compile and load the files just once into a Lisp world, and then save that world in a FEP file. Other users can then save the file on their individual machines. The problem is that since users do not load the system every time they want to use it, they do not get all the latest changes. The patch facility solves this problem, as it 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.

The Lisp Machine 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 via the Command Processor. 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 (*make*) 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; in fact, after a complete recompilation, the old patch files are useless; loading them might even break things.

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 the load 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 the registry's *system version-directory file*, which 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 their own subdirectory in the file system.

In addition to the patch files themselves, the *patch-directory file* keeps track of what minor versions exist for a major version, and what the last major version of a system is. For example, lmfs-37.patch-dir contains a listing of the patches made for major version 37 and a comment on why each patch was made. These files and how to make them are described in this section.

In order to use the patch facility, you must define your system with **defsystem** and declare it as patchable with the **:patchable** option. 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 Lisp 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 Lisp environment away in a FEP file. See the function **load-and-save-patches**, page 246.

13.1 Types of Patch Files

The patch facility maintains several different types of files in the directory associated with your system:

- The system version-directory file
- The patch directory file
- Individual patch files

The system version-directory file and the patch directory file constitute 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.

13.1.1 System Version-directory File

System version information for each patchable system is recorded in a database called the *system version-directory file*. This file associates source and object versions of system files with major versions of the system itself. **make-system** uses this file to determine which versions of the system files to load for the major version you

specified when you called **make-system**. Whenever you run **make-system** with the **:compile**, **:recompile**, or **:update-directory** keywords, **make-system** updates the system version-directory file (or creates it), recording the name, type, and file version number of all files in each version of the system. In addition, it describes the status of the system by associating particular system versions with status keywords, for example, **:released** or **:latest**.

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 **make-system**.

The released version is the fully debugged version intended for general use. To designate a system as the released version use either **set-system-status** (to make the change in memory only) or **make-system** with the **:update-directory** option to make the change in the system version-directory file.

The latest version is the most recently compiled version of the system. The system version-directory file is automatically updated whenever you compile or recompile the system; **make-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.

13.1.2 Patch Directory File

The *patch directory file* 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. **make-system** creates a new patch directory file automatically when you recompile a system or use the **:increment-patch** option.

13.1.3 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 name of the patch source file for major version #37 and minor version #3 of **lmfs**.

13.1.4 Organization of Patch Files

The system version-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 **:pathname-default** 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.

```
          .
          .
(:pathname-default "george: george;")
(:patchable "george:  patch;")
          .
```

When you do not supply the *directory* argument to **:patchable**, then the patches are stored in the directory specified by **:pathname-default**; in the following example this is the logical directory "george: george".

```
          .
          .
(:pathname-default "george: george;")
(:patchable)
          .
```

The source and compiled code patch files for a major system version are kept in their own subdirectory. The patch directory file for a major version resides in the same subdirectory as the patch files. The system version-directory file, which describes all versions of a system, resides in the immediately superior directory.

Patch files are organized in a tree-like structure, as illustrated in the following diagram. Assume that the patches for **george** are stored in the logical directory "george: patch;", which translates into "q:>sys>george>patch>".

```
                                Q
                                |
                               sys
                                |
                              george
                                |
                              patch ______ system version-directory file
                            /       \        for all versions of george
                          /           \
     subdirectory for 1st              subdirectory for nth
     version of george                 version of george______
     |           |     |                 |           |      |
patch directory  patch patch       patch directory  patch  patch
file for 1st     file  file        file for nth     file   file
version of       #1    #n          version of       #1     #n
george                             george
```

13.1.5 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 version-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 LMS would be CHA001 (PDIR) on ITS.

The following tables show the physical file types of the system version-directory file and the patch directory file for various hosts.

Host	*File type of the system version-directory file*
TOPS-20	SYSTEM-DIR
UNIX 4.1	sd
UNIX 4.2	system-dir (also sd for compatibility)
VMS	SPD
ITS	(SDIR)
LMFS	system-dir (also patch-dir or directory for compatibility)
Multics	system-dir

Host	*File type of the patch directory file*
TOPS-20	PATCH-DIR
UNIX 4.1	pd
UNIX 4.2	patch-dir (also pd for compatibility)
VMS	VPD
ITS	(PDIR)
LMFS	patch-dir (also directory for compatibility)
Multics	patch-dir

The format of patch file names varies with the type of file.

- The format of the system version-directory file is some name chosen by the patch facility followed by the appropriate file type and file version number. For example, the system version-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 **si:patch-system-pathname** function to determine the names of your patch files.

si:patch-system-pathname *system type* &rest *args* — *Function*

Returns the logical pathname of a patch file. *system* is the name of the system. *type* is **:system-directory**, **:version-directory**, or **:patch-file**. Specify also any additional *args* required by the type.

Type *Description*

:system-directory

Returns the logical pathname of the system-version directory file for the given system, for example:

```
(si:patch-system-pathname "LMFS"
                          :system-directory)
```

The form returns `#<LOGICAL-PATHNAME "SYS: LMFS; PATCH; LMFS.SYSTEM-DIR.NEWEST">`.

:version-directory

Supplied with a *major-version-number* argument, it returns the logical pathname of that patch directory file for the given system, for example:

```
(si:patch-system-pathname "LMFS"
                          :version-directory 51.)
```

The form returns #<LOGICAL-PATHNAME "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.

```
(si:patch-system-pathname "LMFS"
                          :patch-file 51. 2.
                          :lisp)
```

The form returns #<LOGICAL-PATHNAME "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 `(si:patch-system-pathname "LMFS" :system-directory)`. The form would return #<LMFS-PATHNAME "q:>sys>lmfs>patch>lmfs.system-dir">.

13.2 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 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. View 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 **make-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).

13.2.1 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.

13.2.2 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.

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.

13.2.3 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.

If you mistakenly use the command 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. These commands add only to the current patch buffer and warn you if you try to add code from one system to a patch for another.

13.2.4 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:

Response	*Action*
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".

13.2.5 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:

Response	*Action*
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 View Patches commands to keep track of patches.

13.2.6 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.

13.2.7 View Patches (m-X)

View 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.

View 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.

13.2.8 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 **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.

13.2.9 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.

13.2.10 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.

13.2.11 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 other way except by 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.

13.2.12 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.

13.3 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 to 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.

The Lisp Machine 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 **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; **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 **make-system** to load a particular system, the system-loading facility automatically loads all the patches for that system, using the same options specified in the call to **make-system**.

load-patches &rest *options* *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 **make-system** of a patchable system, **make-system** calls **load-patches** after loading the system. If **make-system** is silent, **load-patches** is silent; if **make-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 (equivalent to using the **:selective** option).

options to **load-patches**, if supplied, can be one or more keywords or system names. The following options are accepted:

system-name — *system-name* is the name of a system (symbol or string) to be brought up to date.

:systems *list* — *list* is a list of names of systems (symbols or strings) to be brought up to date. If this option is not specified, all systems are processed.

:verbose — **:verbose** prints an explanation of what is being done. This is the default.

:selective — For each patch **:selective** displays the patch comment and then asks you whether or not to load the patches. The choices are Y, N, P, or H: yes, no, proceed, or highest. Answering P turns off selective mode for any remaining patches to the current system. H means highest patch number to load. If you do not specify a limit, it loads all patches from the present level for a given system.

:noselective — **:noselective** turns off **:selective**.

:silent — **:silent** turns off both **:selective** and **:verbose**. In **:silent** mode all necessary patches are loaded without printing anything and without querying the user.

:nowarn — **:nowarn** suppresses any warnings generated while a patch file is being loaded, such as that produced when a symbol belonging to one package is being defined by a file belonging to a different package. It also suppresses the subsequent query to the user. **:nowarn** does not imply **:noselective**.

load-and-save-patches &rest *keyword-args* *Function*

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 **:noselective** and any other keywords provided as *keyword-args*. Valid *keyword-args* are:

:verbose — **:verbose** prints an explanation of what is being done. This is the default.

:selective — For each patch **:selective** displays the patch comment and then asks you whether or not to load the patches. The choices are Y, N, P, or H: yes, no, proceed, or highest. Answering P turns off selective mode for any remaining patches to the current system. H means highest patch number to load. If you do not specify a limit, it loads all patches from the present level for a given system.

:silent — **:silent** turns off both **:selective** and **:verbose**. In **:silent** mode all necessary patches are loaded without printing anything and without querying the user.

:nowarn — **:nowarn** suppresses any warnings generated while a patch file is being loaded, such as that produced when a symbol belonging to one package is being defined by a file belonging to a different package. It also suppresses the subsequent query to the user. **:nowarn** does not imply **:noselective**.

If no one is logged in, it logs in anonymously. If any patches have been loaded, **load-and-save-patches** prompts for the name of a FEP file in which to save the world load and then calls **disk-save** to actually save the resulting world load. If no patches have been loaded, it restores network services to their state before **load-and-save-patches** was called, and logs out if it has logged in anonymously.

Call **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.

14. Getting Information About a System

describe-system is a useful general function for finding information about a system, patchable or nonpatchable.

describe-system *system-name* &key *(show-files* **t***)* *(show-transformations* **t***)* *Function*

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 **:show-transformations** is **t** (the default), it displays the transformations required to make the system. Other possible values are **nil** (do not display transformations) and **:ask** (ask the user).

When a Lisp Machine 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 **si: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 **disk-save** in *Installation and Site Operations*.

si: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"
```

si: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 numbers and status associated with them, just as systems do. Symbolics staff assign the release number.

si:get-release-version *Function*

si: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)

print-system-modifications &rest *system-names* *Function*

With no arguments, **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 **print-system-modifications** is called with arguments, only the modifications to *system-names* are listed.

si: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**.

15. Functions That Operate on a System

15.1 Changing the Status of a Patchable System

The major version of a patchable system has a *status* associated with it. This status announces the state, or condition, of the system software — for example, a system can be released, experimental, obsolete, or broken. The status is displayed with the system version, in places such as the system print herald and the **comment** properties in FEP files. Use **set-system-status** to change the status of a system.

set-system-status *system new-status* &optional *major-version only-update-on-disk-p* — *Function*

Changes the status of a patchable system as cached in memory. (Note: To record a change in system status in the system version-directory file, call **make-system** with the **:update-directory** keyword.) Call **set-system-status** manually; you should *not* place the form in patch files.

system — The name of the system.

new-status — One of these defined keywords:

:experimental — The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the **:initial-status** option for **defsystem**.

:released — The system is released for general use. This status produces no extra text in the print herald and the **comment** properties in FEP files.

:obsolete — The system is no longer supported.

:broken — The system was thought incorrectly to have been debugged and was previously assigned **:released** status.

major-version — The number of the major version to be changed; if unsupplied it defaults to the version currently loaded into the Lisp world.

only-update-on-disk-p — If its value is **t**, the patch directory file is updated to show *new-status*, but the running Lisp environment is not modified.

Occasionally, you might want to operate on every file in a defined system. Use **si:map-system-files** and **si:set-system-file-properties** for this purpose.

si:map-system-files *system version function* &rest *args* *Function*

Maps a function over each file in the specified version of the system. The following example deletes every file in version 12 of the **mailer** system.

```
(si:map-system-files 'mailer 12. #'deletef)
```

si:set-system-file-properties *system version* &rest *properties* *Function*

Sets the *properties* of each file in the specified *version* of the *system*. The set of legal properties depends on the file system.

Example: Your directory retains only the two newest versions of each file and discards the rest, but you want to keep every version number of each file in the **mailer** system, major version 10.

```
(si:set-system-file-properties 'mailer 10. :dont-reap t :dont-delete t)
```

PART III.

Debugger

16. Entering the Debugger

When an error condition is signalled and no handlers decide to handle the error, an interactive Debugger is entered to allow you to look around and see what went wrong and to help you continue the program or abort it. This section describes how to use the Debugger and the various debugging facilities.

16.1 Entering the Debugger by Causing an Error

The Debugger is invoked automatically when errors arise during program execution or when you explicitly cause an error, for example, by typing a nonsense symbol name, such as **ahsdgf**, at the Lisp read-eval-print loop.

16.1.1 Error Display

Errors are signalled by the microcode and by Lisp programs (by using **ferror** or related functions). Here is an example of an error:

```
foo

>>Trap: The variable FOO is unbound.

SI:*EVAL:
   Arg 0 (FORM): FOO

s-A, RESUME: Supply a value to use this time as the vaue of FOO
s-B, m-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 Lisp Listener 1
→
```

>> indicates entry to the Debugger. The word immediately following >> shows what caused you to enter the Debugger; most commonly you see `Trap`, `Error`, or `Break`.

`Trap` indicates a microcode error.
`Error` indicates a software error.
`Break` indicates entry by keystroke or the **dbg** function.

The message that follows describes the error in English, in this example, an unbound variable. The next two lines in the example show the stack frame in which the error occurred — the function that was being called and the current value(s) of its argument(s).

The right-facing arrow (→) indicates that the Debugger is waiting for a command. Multiple arrow prompts signal recursive invocations of the Debugger.

16.1.1.1 Debugger Proceed and Restart Options

The Debugger provides options for proceeding from the error or restarting from some prior point. When the Debugger is entered, all *proceed types*, *special commands*, or *restart handlers* available in the error context are assigned to keystrokes with the SUPER modifier, starting with s-A, s-B, and so on, from the most recently established (innermost) to the oldest (outermost). Also, the RESUME key is assigned to the innermost proceed type (or restart handler if there are no proceed types), and the ABORT key is assigned to the innermost restart handler. All these keystroke assignments are displayed when you enter the Debugger or when you type the c-L Debugger command. (See the section "Conditions" in *Reference Guide to Symbolics-Lisp.*)

You can use one of these options or any of the Debugger commands. See the section "How to Use the Debugger", page 259. For details on the Debugger command keys: See the section "Special Keys" in *Reference Guide to Symbolics-Lisp.*

Optionally, you can request that backtrace information appear when you enter the Debugger by setting the variable **dbg:*show-backtrace*** in your init file.

16.2 Entering the Debugger with m-SUSPEND

You can also enter the Debugger explicitly by pressing m-SUSPEND. Adding the CONTROL modifier to this combination has the effect of saying "enter the Debugger immediately". Thus, you can:

- Press m-SUSPEND while the currently running program or read-eval-print loop is reading from the console.
- Press c-m-SUSPEND so that the currently running program enters the Debugger whether or not it is reading from the console.

Note: Pressing the SUSPEND key without the META modifier or just pressing c-SUSPEND enters a read-eval-print loop rather than the Debugger.

16.3 Entering the Debugger with the dbg Function

You can use the **dbg** function in your source code to help detect errors in your programs.

- Insert a call to **dbg** (with no arguments) into your code and then recompile.
- Call **dbg** with an argument of *process* to force a process into the Debugger.

dbg &optional *process* *Function*

Forces *process* into the Debugger so that you can look at its current state.

dbg sets up a restart handler for c-Z, ABORT, and RESUME that exits from the **dbg** function back to the original process. The message for this restart handler is "Allow process to continue". You can use c-T, c-R, c-m-R, and other similar Debugger commands when you enter the Debugger via **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, **errset** and **catch-error** do 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 **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, c-ABORT and c-m-SUSPEND work. Process *Y* starts running the Debugger, using the stream **debug-io**. **debug-io** 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.

For more information: See the special form **break** in *User's Guide to Symbolics Computers*. See the section "Breakpoints" in *Reference Guide to Symbolics-Lisp*.

17. How to Use the Debugger

Once inside the Debugger, you can give a wide variety of commands. With these commands, you can see the arguments for the current stack frame, disassemble its code, return a value for the stack frame, move up and down the stack, and enter the editor to edit function definitions. Press the HELP key or the ? key to display a brief help message or c-HELP for documentation on all of the Debugger commands.

This section describes how to give the commands, and then explains them in approximate order of usefulness.

When the Debugger prompts you with →, you can do one of the following:

- Type a Lisp expression
- Type a Debugger command
- Use the input editor to recall a previous Lisp expression

All Debugger commands are single characters, usually with the CONTROL or META modifiers. The Debugger considers most keys used with a modifier (such as CONTROL or SUPER) to be commands. Most unmodified keys begin a Lisp expression; however, a few keys are commands even without a modifier.

The Debugger and the input editor use some of the same keys for commands. You can enter the input editor at any time by pressing a key that is not a Debugger command, for example, SPACE. Once there, you can type an input editor command that is also a Debugger command.

17.1 Evaluating a Form in the Debugger

When you press a key that is not a command, the Debugger prompts with Eval:, which means that it will evaluate any Lisp expression that you type. The Debugger interprets the Lisp expression as a Lisp form and evaluates it in the context of the function that got the error. That is, all bindings that were in effect at the time of the error will be in effect when your form is evaluated, with certain exceptions explained later in this section. The result of the evaluation is printed, and the Debugger prompts again with an arrow.

If, during the typing of the form, you change your mind and want to get back to the Debugger's command level, press ABORT or c-G; the Debugger responds with an arrow prompt. In fact, you can press ABORT or c-G whenever the Debugger expects typein in order to flush what you are typing and get back to command level.

If a nontrivial error occurs in the evaluation of the Lisp expression, you are thrown into a second Debugger looking at the new error. The Debugger prompts with two arrows (→→) to show that you are inside two Debuggers. You can abort the

computation and get back to the first Debugger by pressing the ABORT key. However, if the error is trivial the abort is done automatically and the original error message is reprinted.

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 nontrivial (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 **y-or-n-p** in *Programming the User Interface*.) If you answer negatively it asks you for another form. To exit the command, just press ABORT or c-G.

17.1.1 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 // 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.

- **rubout-handler** and **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 **step** facility if it had been in use when the error occurred. See the section "**evalhook**", page 289.

- **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.

- base, ibase, and package are checked to insure that they contain legal values. If not, they are set to 8, 8, and **si:pkg-user-package** respectively.

Note that the variable bindings are those in effect at the point of error, *not* those of the current frame being examined.

17.2 Exiting From the Debugger: Abort

The single most useful command is ABORT (or c-Z), which exits from the Debugger and throws out of the computation that got the error. Often you are not interested in using the Debugger at all and just want to get back to the command level in the program you are running; ABORT lets you do this in one character.

The ABORT command returns control to the most recently established restart handler, usually a command or read-eval-print loop. Pressing ABORT multiple times throws you back to successively older read-eval-print or command loops until top level is reached. Pressing c-m-ABORT, on the other hand, always throws you to top level. (Note: c-m-ABORT is not a Debugger command but a system command, which is available from every program.)

Pressing ABORT in the middle of typing a form to be evaluated by the Debugger aborts that form and returns to the Debugger's command level, whereas pressing ABORT as a Debugger command returns out of the Debugger and the erring program to the *previous* command level.

17.3 Debugger Help

Documentation is provided by the HELP or ? command, which displays a very brief explanation of the Debugger. The c-HELP command gives documentation for all of the Debugger commands. If you type c-L or press REFRESH, the Debugger clears the screen, redisplays the error message and the current stack frame, displays a brief backtrace, displays the source file name of a function (when relevant), and lists the special commands that apply to the particular error currently being handled and gives a one-line explanation of each of them.

17.4 Proceeding From the Error in the Debugger: Resume

Often you want to try to proceed from the error. To do this, use the RESUME command. The exact way RESUME works depends on the kind of error that happened. For some errors, there is no standard way to proceed, and RESUME 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.

17.5 Examining the Current Stack Frame in the Debugger

The Debugger knows about a *current stack frame* and has several commands that use it. The initially 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. When the Debugger starts up it shows you this frame in the following format:

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

This means that **foo** was called with two arguments, whose names (in the Lisp source code) are **x** and **y**. The current values of **x** and **y** are **13** and **1** respectively. The Debugger shows the original arguments.

17.6 Examining Stack Frames with Debugger Backtrace Commands

The Debugger provides several commands to allow you to examine the Lisp control stack and to make other frames current than the one that got the error. The control stack (or *regular pdl*) keeps a record of all functions that are currently active. 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* (a backwards trace of the stack) would show all of this information.

Backtraces start at the current frame. Give an argument to specify how many frames to show.

The Debugger has three backtrace commands:

- c-B, which displays a brief backtrace

- m-B, which displays a longer backtrace including the source file name of a function (when relevant)

- c-m-B, which displays the longest backtrace

c-B displays the names of the functions on the stack, starting from the current frame; in the above example it would display:

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

The arrows indicate the direction of calling. A numeric argument specifies how many frames to display.

The m-B command displays a more extensive backtrace including the source file name of a function (when relevant). It indicates the names of the arguments to the functions and their current values; for the example above it might look like:

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

BAR:
   Arg 0 (ADDEND): 13

FOO:
   Arg 0 (FROB): (A B C . D)
```

The c-m-B command displays a verbose backtrace of the stack like the corresponding m-B command, but instead of censoring the stack it additionally displays internal Lisp interpreter frames. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as ***eval**, **prog**, and **cond**, and only show "interesting" functions. The c-m-B command does not skip such frames.

17.7 Debugger Commands for Stack Manipulation

Commands such as c-N and m-N, which are meaningful to repeat, take a prefix numeric argument and repeat that many times. The numeric argument is typed by using c- or m- and the number keys, as in the editor.

The c-N command moves down to the next frame (that is, it changes the current frame to be the frame that called it) and displays the frame in this same format.

c-P or RETURN moves up to the previous frame (that is, the one that this one called) and displays the frame in the same format.

m-< moves to the stack frame where the error occurred (the top or most recent frame) and displays that frame. Use c-P after m-< to go up through **signal**, handlers, and so forth, in turn, until you get to the highest possible frame — the call to the Debugger itself.

m-> goes to the bottom (the oldest frame) and displays that frame.

c-S asks you for a string, and searches the stack for a frame whose executing function's name contains that string. That frame becomes current and is displayed.

m-N moves to the next frame and displays it in full-screen format.

m-P moves to the previous frame and displays it in full-screen format.

The c-m-N command moves down to the next frame and displays it like the corresponding c-N command, but instead of censoring the stack it additionally displays internal Lisp interpreter frames. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as ***eval**, **prog**, and **cond**, and only show "interesting" functions. The c-m-N command does not skip such frames.

The c-m-P command moves up to the previous frame and displays it like the corresponding c-P command, but instead of censoring the stack it additionally displays internal Lisp interpreter frames. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as ***eval**, **prog**, and **cond**, and only show "interesting" functions. The c-m-P command does not skip such frames.

The c-m-U command goes down the stack to the next "interesting" function and makes that the current frame. When running interpreted code, the Debugger tries to skip over frames that belong to functions of the interpreter, such as ***eval**, **prog**, and **cond**, and only show interesting functions.

17.8 Debugger Commands That Call Other Systems

17.8.1 Entering the Editor From the Debugger

c-E puts you into the editor, looking at the source code for the function in the current frame. This is useful when you have found a function that caused the error and needs to be fixed. The editor command c-Z returns to the Debugger, if it is still there.

17.8.2 Sending a Bug Report

c-M sends a bug report. It 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 an extensive backtrace of the stack. It prompts for the number of frames to include in the backtrace. You are expected to supply context information explaining what you were doing when the problem occurred, preferably including a way for the person reading the bug report to make it happen again. The stack trace by itself is not adequate information for debugging. When you type the END key the bug report is transmitted as mail and the window containing the Debugger is reselected.

You can also use normal window-switching commands such as FUNCTION S to switch back and forth between the Debugger and the mail-sending window while composing the bug report. A numeric argument to c-M controls the number of stack frames in the backtrace that have complete information. The current stack frame at the time c-M is typed begins the backtrace, so you might want to type m-< before c-M if you have been examining frames other than the one that got the error.

17.8.3 Entering the Display Debugger

c-m-W calls the Display Debugger, a window-oriented Debugger, which is self-explanatory.

17.9 Debugger Commands for Information Display

Backtraces start at the current frame. Give an argument to specify how many frames to show.

m-L displays the current frame in full-screen format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. If a function **setq**s one of its arguments, m-L shows both the original argument supplied by the caller and the current value of the variable.

c-m-A takes a numeric argument *n*, and displays the value of the *n*th argument of the current frame. The default value for the argument is 0, meaning the first frame.

It leaves * set to the value of the argument, so that you can use the Lisp read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack, so that you can change that argument (by calling **rplacd** on the locative). See the function **dbg:arg**, page 267. See the function **dbg:loc**, page 268.

c-m-L takes a numeric argument *n*, and displays the value of the *n*th local variable of the current frame. The default value for the argument is 0, meaning the first frame. For example, c-m-1 c-m-5 c-m-L displays local argument number 15.

When an error happens in a function that takes an &rest parameter and the actual argument list passed is quite long, you can view the entire &rest argument using c-m-L. For example, if the last argument displayed before the rest argument is arg 3, then c-m-4 c-m-L gets the rest argument. Use c-m-L and add one to the last number it shows you.

c-m-L leaves * set to the value of the argument, so that you can use the Lisp read-eval-print loop to examine it. It also leaves + set to a locative pointing to the

argument on the stack, so that you can change that argument (by calling **rplacd** on the locative).See the function **dbg:arg**, page 267. See the function **dbg:loc**, page 268.

c-m-V takes a numeric argument *n*, and displays the value of the *n*th argument of the frame that was trapped-on-exit. If the frame is not in the process of returning values, the command displays an error message. c-m-V is meaningful only when you are using trap-on-exit (see c-X) and looking at a frame that is about to return. See the function **dbg:val**, page 268.

c-m-F displays the function executing in the current frame. It ignores its numeric argument and does not allow you to change the function. It leaves * set to the value of the argument, so that you can use the Lisp read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack, so that you can change that argument (by calling **rplacd** on the locative). See the function **dbg:fun**, page 268.

c-m-H describes any condition handlers established by the current frame (or its subframes if it is an interpreted function).

c-m-S describes any special-variable bindings in the current frame (or its subframes if it is an interpreted function).

m-S asks for the name of a special variable and displays its value in the binding context of the current frame. It leaves * set to the value that was displayed.

m-I (for *Instance*) helps you examine the values of instance variables in the stack group being debugged. The command prompts you for the name of an instance variable and displays the value of that instance variable, inside the instance that is the value of **self** in the environment of the current frame.

The m-S command can be used to evaluate a special variable in the context of the current frame. This works even for the special variables listed as exceptions (earlier in this section).

c-A displays the argument list of the function in the current frame.

17.10 Debugger Commands That Trap on Frame Exit

If a frame with the trap-on-exit flag set returns or is thrown through, the Debugger is entered. Press RESUME to continue returning or throwing. The ABORT key, however, bypasses the trap-on-exit mechanism. Note that trap on exit also occurs if the frame is thrown through.

c-X toggles the trap-on-exit flag of the current frame and displays its new state.

m-X sets the trap-on-exit flag in the current frame and all its callers.

c-m-X clears the trap-on-exit flag in the current frame and all its callers.

17.11 Debugger Commands for Dynamic Breakpoints and Stepping Through Compiled Code

There are two ways of setting breakpoints in a compiled function. One is to insert a **dbg** call, which enters the debugger when it is executed. The other way is to use the compiled function stepper facility in the debugger. Using this, you can put a breakpoint at an arbitrary instruction in a compiled function.

These commands step through compiled code:

c-sh-S
: Prompts for a function name and a PC (that is, instruction within a function), and puts a breakpoint at that PC. When that PC is reached during execution, the debugger is entered. A numeric argument sets a breakpoint at that PC in the current frame.

m-sh-C
: Clears all breakpoints.

c-sh-L
: Lists breakpoints.

m-sh-S
: Steps through compiled code. Executes one instruction. Note that this steps over function calls, not into the called function.

m-L
: Not a stepping command, but useful to display the contents of the frame while stepping.

17.12 Debugger Functions

The Debugger's command loop lets you type in Lisp forms, which it reads, evaluates, and prints. When you are typing these forms, you can use the following functions to examine or modify the arguments, locals, 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 read-eval-print loop of the Debugger.

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 read-eval-print loop of the Debugger.

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 read-eval-print loop of the Debugger.

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 read-eval-print loop of the Debugger.

17.13 Debugger Variables

The Debugger uses the following variables:

dbg:*frame* *Variable*

Inside the read-eval-print loop of the Debugger, 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 type `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 **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**.

Value	*Meaning*
nil	The Debugger startup message does not include any backtrace information.
t	The Debugger startup message includes a three-element backtrace.

18. Summary of Debugger Commands

c-A	Displays argument list of function in current frame. It displays only the names of the arguments, not their values.
c-m-A	Examines or changes the *n*th argument of the current frame.
c-B	Displays a brief backtrace, including only the names of the functions.
m-B	Displays a more extensive backtrace than c-B, including the names of the arguments to the functions and their current values.
c-m-B	Displays a longer backtrace than c-B and m-B, providing the names of the arguments to the functions and their current values as well as the internal frames of the Lisp interpreter.
RESUME	Attempts to continue execution, if possible.
s-sh-C	Attempts to continue, **setq**ing the unbound variable or otherwise permanently fixing the error.
c-E	Puts you in the editor with the cursor positioned at the source code for the function in the current frame.
c-m-F	Sets * to the function in the current frame.
c-G or ABORT	Quits various Debugger commands; use to escape from typing in a form.
c-m-H	Describes any condition handlers established by the current frame.
m-I	Evaluates an instance variable of the instance that is **self** in the current frame.
c-L, REFRESH	Redisplays error message and current frame.
m-L	Displays full-screen typeout of current frame.
c-m-L	Gets local variable *n*.
c-M	Sends mail to report a bug.
c-N, LINE	Moves to next frame. With argument of *n*, moves down *n* frames.
m-N	Moves to next frame with full-screen typeout. With argument of *n*, moves down *n* frames.
c-m-N	Moves to next frame even if it is "uninteresting". With argument of *n*, moves down *n* frames.
c-P, RETURN	Moves to previous frame. With argument of *n*, moves up *n* frames.
m-P	Moves to previous frame with full-screen typeout. With argument of *n*, moves up *n* frames.

c-m-P	Moves to previous frame even if it is "uninteresting". With argument of *n*, moves up *n* frames.
c-R	Returns from the current frame.
c-m-R	Reinvokes the function in the current frame (throws back to it and starts it over at its beginning).
c-S	Searches for a frame containing a user-specified function.
m-S	Evaluates a special variable in the binding context of the current frame.
c-m-S	Describes any special-variable bindings established by the current frame.
c-T	Throws a value to a tag.
c-m-U	Moves down the stack to the next "interesting" frame.
c-m-V	Gets the *n*th value being returned by the current frame.
c-m-W	Invokes the Display Debugger.
c-X	Toggles the trap-on-exit flag of the current frame.
m-X	Sets the trap-on-exit flag in the current frame and all its callers.
c-m-X	Clears the trap-on-exit flag in the current frame and all its callers.
c-Z, ABORT	Aborts the computation and throws back to the most recent **break** or Debugger, to the program's "command level", or to Lisp top level.
? or HELP	Displays a brief help message.
c-HELP	Displays a detailed help message.
m-<	Goes to top or most recent frame of stack, the stack where the error occurred.
m->	Goes to bottom or oldest frame of stack.
c-sh-S	Prompts for a function name and a PC (that is, instruction within a function), and puts a breakpoint at that PC.
m-sh-C	Clears all breakpoints.
c-sh-L	Lists breakpoints.
m-sh-S	Steps through compiled code. Executes one instruction. Note that this steps over function calls, not into the called function.
c-0—c-m-9	Numeric arguments to the following command are specified by typing a decimal number with the CONTROL and/or META keys held down.

19. Summary of Debugging Aids

Anyone who writes Lisp programs should become familiar with these debugging facilities.

- The *trace* facility provides the ability to perform certain actions at the time a function is called or at the time it returns. The actions can be simple typeout, or more sophisticated debugging functions. See the section "Tracing Function Execution", page 275.

- The *advise* facility is a somewhat similar facility for modifying the behavior of a function. See the section "Advising a Function", page 281.

- The *step* facility allows the evaluation of a form to be intercepted at every step so that the user can examine just what is happening throughout the execution of the form. See the section "Stepping Through an Evaluation", page 287.

- The *evalhook* facility allows you to get at a particular Lisp form whenever the evaluator is called. The step facility uses **evalhook**. See the section "**evalhook**", page 289.

20. 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 value(s).

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. For a table explaining the correspondence between menu options and **trace** options:

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 276. 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 *Reference Guide to Symbolics-Lisp.*) 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.

See the section "Encapsulations" in *Reference Guide to Symbolics-Lisp*.

20.1 Options to trace

The following **trace** options exist:

:break *pred*
Enters a breakpoint 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 breakpoint, the symbol **arglist** is bound to a list of the arguments of the function.

:exitbreak *pred*
This is just like **:break** except that the breakpoint is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the breakpoint, 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 c-X command.)

:step Steps through the function whenever it is called. See the section "Stepping Through an Evaluation", page 287.

: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 is 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.

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

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 *Reference Guide to Symbolics-Lisp*.)

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**.

20.2 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).

20.3 Untracing Function Execution

untrace "e &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.

21. 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 *Reference Guide to Symbolics-Lisp.*

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 *Reference Guide to Symbolics-Lisp.*)

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). The meaning of **:around** advice is explained a couple of sections below.

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 *Reference Guide to Symbolics-Lisp.*

si:advised-functions — *Variable*

A list of all functions that have been advised.

21.1 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.

21.2 :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.

:around advice can access **arglist**, but **values** is not set up until the outermost **:around** advice returns. At that time, it is set to the value returned by the **:around** advice. It is reasonable for the advice to receive the values of the **:do-it** (for example, with **multiple-value-list**) and play with them before returning them (for example, with **values-list**).

:around advice can **return** from the **prog** at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain **:do-it**. Containing two instances of **:do-it** can be useful under peculiar circumstances. If you are careless, however, the original definition might be called twice, but something like the following certainly works reasonably:

```
(if (foo) (+ 5 :do-it) (* 2 :do-it))
```

21.3 Advising One Function Within Another

It is possible to advise the function **foo** only when it is called directly from a specific other function **bar**. You do this by advising the function specifier **(:within bar foo)**. That works by finding all occurrences of **foo** in the definition of **bar** and replacing them with **altered-foo-within-bar**. This can be done even if **bar**'s definition is compiled code. The symbol **altered-foo-within-bar** starts off with the symbol **foo** as its definition; then the symbol **altered-foo-within-bar**, rather than **foo** itself, is advised. The system remembers that **foo** has been replaced inside **bar**, so that if you change the definition of **bar**, or advise it, then the replacement is propagated to the new definition or to the advice. If you remove all the advice on **(:within bar foo)**, so that its definition becomes the symbol **foo** again, then the replacement is unmade and everything returns to its original state.

(grindef bar) prints **foo** where it originally appeared, rather than **altered-foo-within-bar**, so the replacement will not be seen. Instead, **grindef** prints calls to **advise** to describe all the advice that has been put on **foo** or anything else within **bar**.

An alternate way of putting on this sort of advice is to use **advise-within**.

advise-within *within-function function-to-advise class name position* &body *forms* — *Special Form*

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

(advise-within *within-function function-to-advise*
class name position
forms...)

It advises *function-to-advise* only when called directly from the function *within-function*. The other arguments mean the same thing as with **advise**. None of them is evaluated.

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, than 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.

22. Stepping Through an Evaluation

The step facility gives you the ability to follow every step of the evaluation of a 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. See the section "Stepping".

You can enter the stepper in two ways:

- Use the **step** function.
- Use the **:step** option of **trace**.

step *form* — *Function*

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 276. Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be handled by the stepper.

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.

c-N (Next) — Steps to the next thing. The stepper continues until the next thing to print out, and it accepts another command.

SPACE — 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.

If you change the values of these variables, it will work.

c-L Clears the screen and redisplays 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 redisplays 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.

23. 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 **break** and by the Debugger, and **setq**ed 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 only applies 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 **apply**, or on the "mapping" functions. (In Zetalisp, as opposed to Maclisp, it is not necessary to do **(*rset t)** nor **(sstatus evalhook t)**. Also, Maclisp's special-case check for **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 **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 **apply** operations done by **eval**. It does not catch **apply** called in other parts of the interpreter or **apply** or **funcall** operations done by other functions such as **mapcar**. In general, such uses of **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*.

23.1 applyhook

applyhook provides a hook into **apply**, much as **evalhook** provides a hook into **eval**.

applyhook *Variable*

When the value of this variable is not **nil** and **eval** calls **apply**, **applyhook** is bound to **nil** and the function that was its value is applied to two arguments: the function that **eval** gave to **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 IV.

The Inspector

24. Using the Inspector

24.1 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 Display 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 14 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)

24.2 Entering and Leaving the Inspector

You can enter the standalone Inspector via:

- Select Activity *Inspector*
- SELECT I
- [Inspect] in the System menu
- The Inspect command, which inspects its argument, if any

```
                              More above                       Exit
  #<Package GLOBAL 20315016>                                   Return
  1130                                                         Modify
  "GLOBAL"                                                     DeCache
  SI:PKG-NEW-SYMBOL-EXTERNAL-ONLY                              Clear
→ :SOURCE-FILE-NAME                                            Set \
                              More below
                              Top of object
SI:PKG-NEW-SYMBOL-EXTERNAL-ONLY
Value is unbound
Function is #'SI:PKG-NEW-SYMBOL-EXTERNAL-ONLY
Property list: (:SOURCE-FILE-NAME #<LOGICAL-PATHNAME "SYS: SYS; PACKAGE">)
Package: #<Package SYSTEM-INTERNALS 20043232>

                              Bottom of object
                              Top of object
:SOURCE-FILE-NAME
Value is :SOURCE-FILE-NAME
Function is unbound
Property list: NIL
Package: #<Package KEYWORD 20333021>

                              Bottom of object
                              Top of object
#<LOGICAL-PATHNAME "SYS: SYS; PACKAGE">
An instance of FS:LOGICAL-PATHNAME.  #<Message handler for FS:LOGICAL-PATHNAME>

FS:HOST:                  #<LOGICAL-HOST SYS>
FS:DEVICE:                :UNSPECIFIC
FS:DIRECTORY:             ("SYS")
FS:NAME:                  "PACKAGE"
FS:TYPE:                  NIL
FS:VERSION:               NIL
SI:PROPERTY-LIST:         #<LMFS-PATHNAME "Q:>sys>sys>package">
FS:STRING-FOR-PRINTING:   "SYS: SYS; PACKAGE"

                              Bottom of object
L: Inspect the indicated object.  M: Remove the indicated object.
12/07/83 19:10:49 sr          USER:     Tyi___          Console idle 10 minutes
```

Figure 14.   The Inspector.

- 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.

24.3 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.

24.4 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.

24.5 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 **inspect**.

[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 redisplays 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.

24.6 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).

24.6.1 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 **car** or atom 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.

24.7 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.

24.8 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.

PART V.

The Peek Program

25. Peek

You start up Peek by pressing SELECT P or by using the Select Activity Peek command.

The Peek program gives a dynamic display of various kinds of system status. When you start up a 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 keyboards keys as listed in the Help message also switches Peek to the mode associated with that key. The Help message is a Peek mode and Peek starts out in the Help 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 inter-update time interval 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 overruns the available display area. See the section "Scrolling".

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.

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.

peek &optional (*character* **(quote tv:p)**) *Function*

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, **peek** starts out by explaining what its modes are.

PART VI.

The Compiler

26. Introduction to the Compiler

The purpose of the Lisp compiler is to convert interpreted Lisp functions into programs in the Symbolics Lisp Machine'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, undeclared variables, and the like. 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 compiled rather than debugging them interpreted and compiling them after they work.

26.1 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 Lisp environment. Compiling an interpreted function in a 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 **compiler: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. The purpose of these commands is to produce a translated version that does the same thing as the original except that the functions are compiled.

- Invoke **make-system** or Compile System at the Command Processor prompt to compile and load large programs, usually consisting of many files.

27. Structure of the Compiler

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

- The stream compiler
- The function compiler
- The **bin** 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 5. 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.

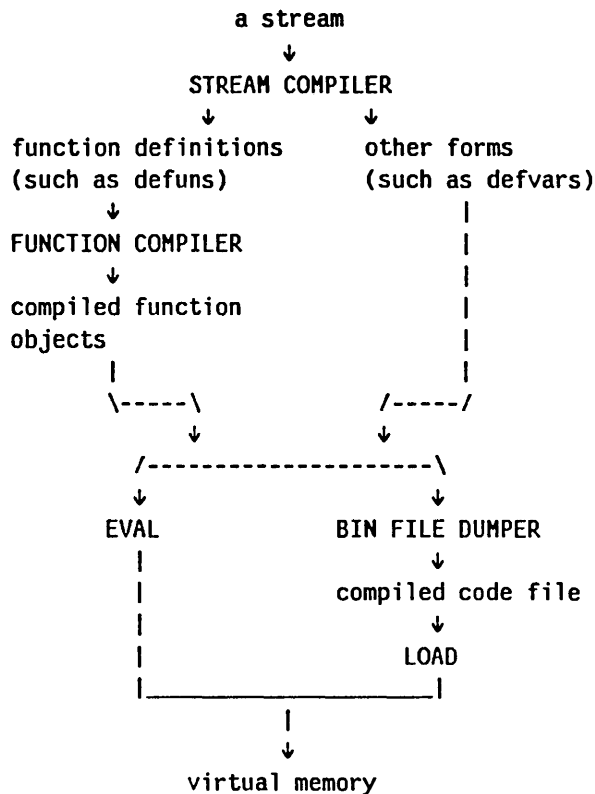
The function compiler takes a Lisp function and translates it from Lisp expressions into machine instructions. Its job includes expanding macros, performing optimizations, recognizing special forms, and recognizing calls to functions that have corresponding machine instructions. The function compiler 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 (**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 meaningful and available:

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

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



The Lisp Machine tools you use to invoke compilation determine the path through the diagram. For example, suppose you run the **compiler: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 and then passes the resulting compiled function objects to the **bin** file dumper. Other forms are passed directly to the **bin** file dumper (right-hand side of the diagram) without being processed through the function compiler. All output 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 (m-X); the output from the stream compiler and function compiler is evaluated immediately (left-hand side of the diagram). 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.

27.1 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 the table below.

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 the table. 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 Lisp Machine 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.

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 *Reference Guide to Symbolics-lisp.*

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

> 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 *Reference Guide to Symbolics-lisp.*

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 *Reference Guide to Symbolics-lisp.*

Variable Definitions, such as **(defvar...)**, **(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 *Reference Guide to Symbolics-lisp.*

(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

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

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

(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 the intepreter, 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**.

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

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

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 **(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 **special**, **unspecial**, ***expr**, ***lexpr**, and ***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**.

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.

27.1.1 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: 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))
```

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.

27.2 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.
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.

27.3 bin File Dumper

The **bin** 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 file (called 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 25.

By loading the compiled code file (using the **load** function), the objects represented in the file are created in your Lisp world.

27.4 Compiler Tools and Their Differences

27.4.1 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 compiles 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.)

27.4.2 Tools for Compiling Files

Compiling a source file, using Compile File (m-X) or **compiler: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 **compiler:compile-file-load**, or you can load the file separately into your Lisp world by using **load** or Load File (m-X).

compiler:compile-file *infile* &optional *outfile in-package dont-set-default-p* *Function*

The file *infile* is given to the compiler, and the output of the compiler is written to a file whose name is *infile* with a canonical file type of **:bin**. *outfile*, if supplied, lets you change where the output is written. *dont-set-default-p* suppresses the changing of the default file name to *infile*, which normally occurs.

The purpose of **compiler: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. **compiler: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.)

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

compiler:compile-file-load *infile* &optional *outfile in-package dont-set-default-p* *Function*

compiler:compile-file-load compiles a file and then loads the resulting compiled code file. The file *infile* is given to the compiler, and the output of the compiler is written to a file whose name is *infile* with a canonical file type of **:bin**. *outfile*, if supplied, lets you change where the output is written. *dont-set-default-p* suppresses the changing of the default file name to *infile*, which normally occurs.

Compile File (m-X)

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.

27.4.2.1 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.

Host type	*File type of source file*	*File type of compiled code file*
Lisp Machine	lisp	bin
Multics	lisp	bin
TOPS-20	LISP, LSP	BIN
UNIX	l, lisp	bn, bin
VAX/VMS	LSP	BIN

27.4.3 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 *Reference Guide to Symbolics-lisp*.

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 *Reference Guide to Symbolics-lisp*.

Although all these methods call the compiler and produce compiled function objects, they are by no means equivalent. For example, using **compiler: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.

28. Compiler Warnings Database

Compiler warnings are kept in an internal database. Several functions and editor 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 are printed out as well as stored in the database. If the value of the special variable **suppress-compiler-warnings** is not **nil**, warnings are not printed, although they are still stored in the database.

The database has a printed representation. **print-compiler-warnings** produces this printed representation from the database, and **compiler:load-compile-warnings** updates the database from a saved printed representation.

print-compiler-warnings &optional *files* (*stream* **standard-output**) *file-node-message function-node-message anonymous-function-node-message* — *Function*

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 this 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 **make-system** with the **:batch** option. The default type for such files is 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.

29. Controlling Compiler Warnings

29.1 Compiler Style Warnings

The compiler performs style checking on all forms. 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 **explode**, since **inhibit-style-warnings** applies only to the top level of the form inside it, in this case, to the **setq**.

Right:

```
(inhibit-style-warnings (setq bar (explode foo)))
```

The following code, on the other hand, does *not* warn that **explode** is an obsolete function:

Wrong:

```
(setq bar (inhibit-style-warnings (explode foo)))
```

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* **'defun**) *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*, 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 three definition-types: **defun, defflavor**, and **defstruct**.

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.")
```

29.2 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 got 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 get 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**).

***expr**, ***lexpr**, and ***fexpr** are the Maclisp equivalents of **compiler:function-defined.**

***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.

***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.

***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* **compiler:default-warning-function**) *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.

29.2.1 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 two things.

- 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)
  ...)
```

30. Compiler Source-level Optimizers

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.

compiler:add-optimizer *target-function optimizer-name* &rest *optimized-into* — *Special Form*

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.

31. Files That Maclisp Must Compile

Certain programs are intended to be run both in Maclisp and in Zetalisp. 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 Zetalisp 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 Zetalisp. 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 **if-for-lispm** and **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:

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 Zetalisp. If the Zetalisp interpreter sees this it evaluates *forms* (the macro expands into *forms*).

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 Zetalisp interpreter sees this it ignores it (the macro expands into **nil**).

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.

if-in-lispm &rest *forms* *Macro*

In Zetalisp, **(if-in-lispm** *forms***)** causes *forms* to be evaluated; in Maclisp, *forms* is ignored.

if-in-maclisp &rest *forms* *Macro*

In Maclisp, **(if-in-maclisp** *forms***)** causes *forms* to be evaluated; in Zetalisp, *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 the Symbolics Lisp Machine. 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 the Symbolics Lisp Machine and to **nil** when evaluated in Maclisp.

32. 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 printed representations. 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 **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 real definition writes a binary file 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 **user** package. 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.

Index

#

#M sharp-sign reader macro 329
#Q sharp-sign reader macro 329

1

One Window (c-X 1) Zmacs command 65

2

Two Windows (c-X 2) Zmacs command 65

3

View Two Windows (c-X 3) Zmacs command 65

4

Modified Two Windows (c-X 4) Zmacs command 65

;

Set Comment Column (c-X ;) Zmacs command 23

A

ABORT 98
Exiting From the Debugger: Abort 261
ABORT Debugger command 264, 271
sys: **abort** flavor 129
Abort Patch (m-X) 243
Getting Information About a System 249
Finding Out About Existing Code 35
Send mail about patch 242
Summary of Compiler Actions on Code in a Zmacs Buffer 71
Active patches 238, 242
Display status of active processes 303
Select Activity command 9
Adding New Keywords to **make-system** 227
Adding New Options to **defsystem** 214
compiler: **add-optimizer** special form 327
Add Patch Changed Definitions (m-X) 241
Add Patch Changed Definitions of Buffer (m-X) 241
Add Patch (m-X) 240
Add region to patch file 238
:around Advice 284
Designing the Advice 283
Advice to functions 281
si: **advise-1** function 283
si: **advised-functions** variable 283
advise special form 281
advise-within special form 285
Advising a Function 281
Advising One Function Within Another 285

[Break after] **trace** menu item 92
[Cond after] **trace** menu item 92
[Cond break after] **trace** menu item 92, 98
[Print after] **trace** menu item 92
Summary of Debugging Aids 273
Programming Aids for Flavors and Windows 141
Aligning code 26
Aligning Code: Program Development Tools and Techniques 26
Select All Buffers As Tag Table (m-X) Zmacs command 57
Macro Expand Expression All (m-X) Zmacs command 100
Clearing the Trap-on-exit Flag for the Current and All Outer Frames 266
Setting the Trap-on-exit Flag for the Current and All Outer Frames 266
Anonymous module 199
Advising One Function Within Another 285
:any-tyi method of **tv:any-tyi-mixin** 129
:any-tyi method of **tv: any-tyi-mixin** 129
tv: any-tyi-mixin flavor 129
applyhook 290
applyhook function 291
applyhook variable 291
apropos function 38
Function Apropos (m-X) Zmacs command 41
Display status of window area 303
Display status of areas 303
dbg: arg function 267
Quick Arglist (c-sh-A) Zmacs command 43
arglist function 43
arglist variable 98, 276
:arg option for **trace** 92
:arg option to **trace** 278
:argpdl option for **trace** 92
:argpdl option to **trace** 277
[ARGPDL] **trace** menu item 92
:argpdl trace Option 277
Argument lists 43
Argument Lists: Program Development Tools and Techniques 43
Ignored arguments 324
patch-atom argument to **:patchable** option for **defsystem** 236
:around Advice 284
Bit-save array 116, 129
Inspecting an array 299
Arrays in compiled code files 331
Mouse cursor as an arrow 297
The Basic Arrow Window 116
The Arrow Window: Interaction, Processes, and the Mouse 129
Atom Word Mode (m-X) Zmacs command 12
Reparse Attribute List (m-X) Zmacs command 10
Update Attribute List (m-X) Zmacs command 10
Attribute lists (in files) 10
File Attribute Lists: Program Development Tools and Techniques 10
Attributes (of buffers) 10
[Attributes] System menu item 141
Auto Fill Mode (m-X) Zmacs command 12

B

Set Backspace (m-X) Zmacs command 10
Backtrace 264, 271
Examining Stack Frames with Debugger Backtrace Commands 262
Backtrace information 268
Backtrace of the call stack 268
Backward Kill Sexp (c-m-RUBOUT) Zmacs command 60
Balancing Parentheses 26
Balancing Parentheses: Program Development Tools and Techniques 26
Base 10
Set Base (m-X) Zmacs command 10
The Basic Arrow Window 116
:send-command method of **lgp:** **basic-lgp-stream** 124
:send-coordinates method of **lgp:** **basic-lgp-stream** 124
lgp: **basic-lgp-stream** flavor 124
si: ***batch-mode-p*** variable 229
:batch option for **make-system** 79, 222
Beep (c-G) Zmacs command 59
[Break before] **trace** menu item 92
[Cond before] **trace** menu item 92
[Cond break before] **trace** menu item 92, 98
[Print before] **trace** menu item 92
Before You Begin: Program Development Tools and Techniques 7
Before You Begin: Program Development Tools and Techniques 7
Rebound Variable Bindings During Evaluation 260
bin file dumper 309, 316
bin file type 318
Bit-save array 116, 129
:blinker-p init option for **tv:sheet** 116
Blinkers 133
:both option for **trace** 92
:both option to **trace** 278
[Break after] **trace** menu item 92
[Cond break after] **trace** menu item 92, 98
[Break before] **trace** menu item 92
[Cond break before] **trace** menu item 92, 98
BREAK Inspector command 300
breakon function 98
:break option for **trace** 92, 98
:break option to **trace** 276
Breakpoints 98
Debugger Commands for Dynamic Breakpoints and Stepping Through Compiled Code 267
Breakpoints: Program Development Tools and Techniques 98
break special form 98
:break trace Option 276
Brief Documentation (c-sh-D) Zmacs command 39, 42
:broken system status 251
Compiling Code in a Zmacs Buffer 70
Summary of Compiler Actions on Code in a Zmacs Buffer 71
Select Previous Buffer (c-m-L) Zmacs command 59
Select Buffer (c-X B) Zmacs command 59
Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 70
Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 75
Add Patch Changed Definitions of Buffer (m-X) 241
Compile Buffer (m-X) Zmacs command 70

Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
Evaluate Buffer (m-X) Zmacs command 75
Evaluate And Replace Into Buffer (m-X) Zmacs command 75
Evaluate Into Buffer (m-X) Zmacs command 75
Insert Buffer (m-X) Zmacs command 63
List Changed Definitions Of Buffer (m-X) Zmacs command 56
Copying buffers 63
Multiple buffers 65
Attributes (of buffers) 10
Copying Buffers and Files: Program Development Tools and Techniques 63
Select All Buffers As Tag Table (m-X) Zmacs command 57
Multiple Buffers: Program Development Tools and Techniques 65
bug function 264
Bug mail 264, 271
Sending a Bug Report in the Debugger 264
Bug reports 264, 271
:bug-reports Option for **defsystem** 196
Select Buffer (c-X B) Zmacs command 59

C

Indent For Comment (c-; or m-;) Zmacs command 23
c-? Zmacs minibuffer command 8
c-A Debugger command 80, 271
Calculation Module for the Sample Program 147
Callers 44
Edit Callers (m-X) Zmacs command 44
List Callers (m-X) Zmacs command 38, 44
Multiple Edit Callers (m-X) Zmacs command 44
Multiple List Callers (m-X) Zmacs command 44
Callers: Program Development Tools and Techniques 44
Call Last Kbd Macro (c-X E) Zmacs command 64
Debugger Commands That Call Other Systems 264
Backtrace of the call stack 268
:case method combination 133
Entering the Debugger by Causing an Error 255
c-B Debugger command 264, 271
c-B Stepper command 94
Display Directory (c-X c-D) Zmacs command 45
c-E Debugger command 80, 264, 271
c-E Stepper command 94
Find File (c-X c-F) Zmacs command 9
Beep (c-G) Zmacs command 59
Changed code 56
Identifying Changed Code: Program Development Tools and Techniques 56
Add Patch Changed Definitions (m-X) 241
Compile Changed Definitions (m-X) Zmacs command 70
Edit Changed Definitions (m-X) Zmacs command 56
Evaluate Changed Definitions (m-X) Zmacs command 75
List Changed Definitions (m-X) Zmacs command 56
Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 70
Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 75
Add Patch Changed Definitions of Buffer (m-X) 241
Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
List Changed Definitions Of Buffer (m-X) Zmacs command 56

:change-of-size-or-margins method of **tv:sheet** 116
Changing the Status of a Patchable System 251
Special Characters Recognized by the Inspector 300
c-HELP Debugger command 80, 264, 271
:function option for **tv:** **choose-variable-values** 133
tv: **choose-variable-values** function 126, 133
Choose-variable-values window 126, 129
c-L Debugger command 80, 264, 271
Clearing the Trap-on-exit Flag for the Current and All Outer Frames 266
[Clear] Inspector menu item 298
Mouse clicks 129
Inspecting a closure 299
Kill Comment (c-m-;) Zmacs command 23
c-m-A Debugger command 264, 271
c-m-B Debugger command 264, 271
c-M Debugger command 264, 271
c-m-F Debugger command 264, 271
c-m-H Debugger command 264, 271
Mark Definition (c-m-H) Zmacs command 60
Kill Sexp (c-m-K) Zmacs command 60
c-m-L Debugger command 264, 271
Select Previous Buffer (c-m-L) Zmacs command 59
c-m-N Debugger command 264, 271
c-m-P Debugger command 264, 271
Indent Sexp (c-m-Q) Zmacs command 26
c-m-R Debugger command 80, 264, 271
Backward Kill Sexp (c-m-RUBOUT) Zmacs command 60
c-m-S Debugger command 264, 271
Move To Previous Point (c-m-SPACE) Zmacs command 59
c-m-SUSPEND 80
Indent For Lisp (TAB or c-m-TAB) Zmacs command 26
c-m-U Debugger command 264, 271
c-m-V Debugger command 264, 271
Scroll Other Window (c-m-V) Zmacs command 65
c-m-W Debugger command 80, 264, 271
c-m-X Debugger command 264, 271
c-m-Y input editor command 77
Indent Region (c-m-\) Zmacs command 26
c-N Debugger command 80, 264, 271
c-N Stepper command 94
Aligning code 26
Changed code 56
Compiling code 69, 70
Compiling Lisp Code 70
Debugger Commands for Dynamic Breakpoints and Stepping Through Compiled Code 267
Evaluating code 69, 75, 94
Evaluating Lisp Code 75
Finding Out About Existing Code 35
Writing and Editing Code 7
Examining a Compiled Code File 300
Arrays in compiled code files 331
Compiled code objects in compiled code files 331
File Types of Lisp Source and Compiled Code Files 318
Instances in compiled code files 331
Lists in compiled code files 331
Numbers in compiled code files 331
Putting Data in Compiled Code Files 331
Symbols in compiled code files 331
Tools for Compiling Code From the Editor Into Your World 316
Compiling Code in a Zmacs Buffer 70
Summary of Compiler Actions on Code in a Zmacs Buffer 71

Inspecting a compiled code object 299
Compiled code objects in compiled code files 331
Aligning Code: Program Development Tools and Techniques 26
Commenting Out Code: Program Development Tools and Techniques 83
Editing Code: Program Development Tools and Techniques 56
Identifying Changed Code: Program Development Tools and Techniques 56
Set Comment Column (c-X ;) Zmacs command 23
Set Fill Column (c-X F) Zmacs command 12
:case method combination 133
:daemon method combination 113, 116, 121
Edit Combined Methods (m-X) Zmacs command 142
List Combined Methods (m-X) Zmacs command 142
ABORT Debugger command 264, 271
Atom Word Mode (m-X) Zmacs command 12
Auto Fill Mode (m-X) Zmacs command 12
Backward Kill Sexp (c-m-RUBOUT) Zmacs command 60
Beep (c-G) Zmacs command 59
BREAK Inspector command 300
Brief Documentation (c-sh-D) Zmacs command 39, 42
c-? Zmacs minibuffer command 8
c-A Debugger command 80, 271
Call Last Kbd Macro (c-X E) Zmacs command 64
c-B Debugger command 264, 271
c-B Stepper command 94
c-E Debugger command 80, 264, 271
c-E Stepper command 94
c-HELP Debugger command 80, 264, 271
c-L Debugger command 80, 264, 271
c-m-A Debugger command 264, 271
c-m-B Debugger command 264, 271
c-M Debugger command 264, 271
c-m-F Debugger command 264, 271
c-m-H Debugger command 264, 271
c-m-L Debugger command 264, 271
c-m-N Debugger command 264, 271
c-m-P Debugger command 264, 271
c-m-R Debugger command 80, 264, 271
c-m-S Debugger command 264, 271
c-m-U Debugger command 264, 271
c-m-V Debugger command 264, 271
c-m-W Debugger command 80, 264, 271
c-m-X Debugger command 264, 271
c-m-Y input editor command 77
c-N Debugger command 80, 264, 271
c-N Stepper command 94
Compile Buffer (m-X) Zmacs command 70
Compile Changed Definitions (m-X) Zmacs command 70
Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 70
Compile File (m-X) Zmacs command 73
Compile Region (c-sh-C) Zmacs command 70
Compile Region (m-X) Zmacs command 316
Compiler Warnings (m-X) Zmacs command 79, 321
Compile System command 217
COMPLETE Zmacs minibuffer command 8
c-P Debugger command 80, 264, 271
c-R Debugger command 80, 264, 271
c-S Debugger command 264, 271
c-sh-C Zmacs command 316

c-sh-P Debugger command 264, 271
c-T Debugger command 264, 271
c-U Stepper command 94
c-X Debugger command 271
c-X Stepper command 94
c-Z Inspector command 300
Deinstall Macro (m-X) Zmacs command 64
Describe Flavor (m-X) Zmacs command 141
Describe Variable At Point (c-sh-V) Zmacs command 39
Dired (m-X) Zmacs command 45
Disassemble (m-X) Zmacs command 104
Display Directory (c-X c-D) Zmacs command 45
Down Comment Line (m-N) Zmacs command 23
Edit Callers (m-X) Zmacs command 44
Edit Changed Definitions (m-X) Zmacs command 56
Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
Edit Combined Methods (m-X) Zmacs command 142
Edit Compiler Warnings (m-X) Zmacs command 79, 321
Edit Definition (m-.) Zmacs command 40, 141, 142
Edit File Warnings (m-X) Zmacs command 321
Edit Methods (m-X) Zmacs command 65, 142
Electric Shift Lock Mode (m-X) Zmacs command 12
END Zmacs minibuffer command 8
Evaluate And Replace Into Buffer (m-X) Zmacs command 75
Evaluate Buffer (m-X) Zmacs command 75
Evaluate Changed Definitions (m-X) Zmacs command 75
Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 75
Evaluate Into Buffer (m-X) Zmacs command 75
Evaluate Minibuffer (m-ESCAPE) Zmacs command 75
Evaluate Region (c-sh-E) Zmacs command 75
Fill Long Comment (m-X) Zmacs command 23
Find File (c-X c-F) Zmacs command 9
Find Unbalanced Parentheses (m-X) Zmacs command 26
Function Apropos (m-X) Zmacs command 41
HELP Debugger command 264, 271
HELP Stepper command 94
HELP Zmacs command 7, 64
HELP Zmacs minibuffer command 8
Incremental Search (c-S) Zmacs command 57
Indent For Comment (c-; or m-;) Zmacs command 23
Indent For Lisp (TAB or c-m-TAB) Zmacs command 26
Indent New Comment Line (m-LINE) Zmacs command 23
Indent New Line (LINE) Zmacs command 26
Indent Region (c-m-\) Zmacs command 26
Indent Sexp (c-m-Q) Zmacs command 26
Insert Buffer (m-X) Zmacs command 63
Insert File (m-X) Zmacs command 63
Inspect command 295
Install Macro (m-X) Zmacs command 64
Install Mouse Macro (m-X) Zmacs command 64
Jump To Saved Position (c-X J) Zmacs command 59
Kill Comment (c-m-;) Zmacs command 23
Kill Sexp (c-m-K) Zmacs command 60
Lisp Mode (m-X) Zmacs command 12
List Callers (m-X) Zmacs command 38, 44
List Changed Definitions (m-X) Zmacs command 56
List Changed Definitions Of Buffer (m-X) Zmacs command 56
List Combined Methods (m-X) Zmacs command 142
List Matching Lines (m-X) Zmacs command 57
List Matching Symbols (m-X) Zmacs command 38
List Methods (m-X) Zmacs command 142
Load Compiler Warnings (m-X) Zmacs command 79, 321

Load File (m-X) Zmacs command 73
Load System command 217
Long Documentation (m-sh-D) Zmacs command 39, 42
m-< Debugger command 264, 271
m-> Debugger command 264, 271
Macro Expand Expression All (m-X) Zmacs command 100
Macro Expand Expression (c-sh-M) Zmacs command 100
Mark Definition (c-m-H) Zmacs command 60
Mark Whole (c-X H) Zmacs command 63
m-B Debugger command 80, 264, 271
m-I Debugger command 264, 267, 271
m-L Debugger command 80, 264, 271
m-N Debugger command 264, 271
Modified Two Windows (c-X 4) Zmacs command 65
Move To Previous Point (c-m-SPACE) Zmacs command 59
m-P Debugger command 264, 271
m-S Debugger command 264, 271
Multiple Edit Callers (m-X) Zmacs command 44
Multiple List Callers (m-X) Zmacs command 44
m-X Debugger command 264, 271
m-Y Input editor command 77
Name Last Kbd Macro (m-X) Zmacs command 64
One Window (c-X 1) Zmacs command 65
Open Get Register (c-X G) Zmacs command 63
Other Window (c-X O) Zmacs command 65
Print Modifications (m-X) Zmacs command 56
Push Pop Point Explicit (m-SPACE) Zmacs command 59
Put Register (c-X X) Zmacs command 63
Query Replace (m-%) Zmacs command 57
Quick Arglist (c-sh-A) Zmacs command 43
Quit (c-Z) Zmacs command 80
Reparse Attribute List (m-X) Zmacs command 10
Replace (c-%) Zmacs command 57
RESUME Debugger command 264, 271
RETURN Zmacs minibuffer command 8
Reverse Search (c-R) Zmacs command 57
Save Position (c-X S) Zmacs command 59
Save Region (m-W) Zmacs command 60
Scroll Other Window (c-m-V) Zmacs command 65
Select Activity command 9
Select All Buffers As Tag Table (m-X) Zmacs command 57
Select Buffer (c-X B) Zmacs command 59
Select Previous Buffer (c-m-L) Zmacs command 59
Select System As Tag Table (m-X) Zmacs command 57
Set Backspace (m-X) Zmacs command 10
Set Base (m-X) Zmacs command 10
Set Comment Column (c-X ;) Zmacs command 23
Set Fill Column (c-X F) Zmacs command 12
Set Fonts (m-X) Zmacs command 10
Set Key (m-X) Zmacs command 64
Set Lowercase (m-X) Zmacs command 10
Set Nofill (m-X) Zmacs command 10
Set Package (m-X) Zmacs command 10
Set Patch File (m-X) Zmacs command 10
Set Pop Mark (c-SPACE) Zmacs command 59
Set sleep time between updates Peek command 303
Set Tab Width (m-X) Zmacs command 10
Set Vsp (m-X) Zmacs command 10
Source Compare Merge (m-X) Zmacs command 56
Source Compare (m-X) Zmacs command 56
SPACE Stepper command 94
SPACE Zmacs minibuffer command 8
Split Screen (m-X) Zmacs command 65

s-sh-C Debugger	command 264, 271
Swap Point And Mark (c-X c-X) Zmacs	command 59
Tags Query Replace (m-X) Zmacs	command 57
Tags Search (m-X) Zmacs	command 57
Trace (m-X) Zmacs	command 92, 94
Two Windows (c-X 2) Zmacs	command 65
Up Comment Line (m-P) Zmacs	command 23
Update Attribute List (m-X) Zmacs	command 10
View Directory (m-X) Zmacs	command 45
View Two Windows (c-X 3) Zmacs	command 65
Where Is Symbol (m-X) Zmacs	command 38
Yank (c-Y) Zmacs	command 60
Yank Pop (m-Y) Zmacs	command 60
↑r Dired (c-X D) Zmacs	command 45
ESCAPE Inspector	command 300
Debugger	commands 271
Debugger special	commands 80
Examining Stack Frames with Debugger Backtrace	Commands 262
Inspector	commands 298
Summary of Debugger	Commands 271
Debugger	Commands for Dynamic Breakpoints and Stepping Through Compiled Code 267
Debugger	Commands for Information Display 265
Debugger	Commands for Stack Manipulation 263
Debugger	Commands That Call Other Systems 264
Debugger	Commands That Trap on Frame Exit 266
Indent For	Comment (c-; or m-;) Zmacs command 23
Kill	Comment (c-m-;) Zmacs command 23
Set	Comment Column (c-X ;) Zmacs command 23
	Commenting Out Code: Program Development Tools and Techniques 83
Indent New	Comment Line (m-LINE) Zmacs command 23
Down	Comment Line (m-N) Zmacs command 23
Up	Comment Line (m-P) Zmacs command 23
Fill Long	Comment (m-X) Zmacs command 23
	Comments 23
	Comments: Program Development Tools and Techniques 23
Source	Compare Merge (m-X) Zmacs command 56
Source	Compare (m-X) Zmacs command 56
Files That Maclisp Must	Compile 329
	Compile Buffer (m-X) Zmacs command 70
	Compile Changed Definitions (m-X) Zmacs command 70
	Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 70
Debugger Commands for Dynamic Breakpoints and Stepping Through	Compiled Code 267
Examining a	Compiled Code File 300
Arrays in	compiled code files 331
Compiled code objects in	compiled code files 331
File Types of Lisp Source and	Compiled Code Files 318
Instances in	compiled code files 331
Lists in	compiled code files 331
Numbers in	compiled code files 331
Putting Data in	Compiled Code Files 331
Symbols in	compiled code files 331
Inspecting a	compiled code object 299
	Compiled code objects in compiled code files 331
	Compiled function object 309
	Compiled functions 69
compiler:	**compile-file** function 73, 307, 317
compiler:	**compile-file-load** function 73, 318

Compile File (m-X) Zmacs command 73
compile-flavor-methods macro 133
compile function 307, 319
Compile-like transformations 214
:compile-load compound transformation 201
:compile-load-init Transformation of **defsystem** 211
:compile-load Transformation of **defsystem** 210
:compile Option for **make-system** 222
Function compiler 309, 315
How to Invoke the Compiler 307
Introduction to the Compiler 307
Optimizer feature of the compiler 327
Stream compiler 309
Structure of the Compiler 309
The Compiler 305
Summary of Compiler Actions on Code in a Zmacs Buffer 71
compiler:add-optimizer special form 327
compiler:compile-file function 73, 307, 317
compiler:compile-file-load function 73, 318
Compile Region 316
Compile Region (c-sh-C) Zmacs command 70
Compile Region (m-X) Zmacs command 316
Specifying compiler environments 329
compiler:file-declaration function 325
compiler:file-declare function 325
compiler:function-defined function 325
compiler:function-referenced function 325
compiler:functions-defined variable 324
compiler:functions-referenced variable 325
How the Stream Compiler Handles Top-level Forms 310
compiler-let 310
compiler:load-compiler-warnings function 321
compiler:make-message-obsolete special form 324
compiler:make-obsolete special form 323
Compiler Source-level Optimizers 327
Compiler Style Warnings 323
Compiler Tools and Their Differences 316
compiler:top-level-form property 315
Compiler variables 324
Compiler warnings 65, 70, 79
Controlling Compiler Warnings 323
Compiler Warnings Database 321
Print compiler warnings database 321
Update compiler warnings database 321
The Compiler Warnings Database: Program Development Tools and Techniques 79
Compiler Warnings (m-X) Zmacs command 79, 321
Edit Compiler Warnings (m-X) Zmacs command 79, 321
Load Compiler Warnings (m-X) Zmacs command 79, 321
Compile System command 217
:compile Transformation of **defsystem** 209
Compiling and Evaluating Lisp 69
Compiling and Loading a File 73
Compiling code 69, 70
Tools for Compiling Code From the Editor Into Your World 316
Compiling Code in a Zmacs Buffer 70
Tools for Compiling Files 317
Compiling Lisp Code 70
Tools for Compiling Single Functions 318
COMPLETE Zmacs minibuffer command 8
Completion 8
Completion: Program Development Tools and Techniques 8

Complex transformations 214
:component-systems Option for **defsystem** 197
:compile-load compound transformation 201
[Cond after] **trace** menu item 92
[Cond before] **trace** menu item 92
[Cond break after] **trace** menu item 92, 98
[Cond break before] **trace** menu item 92, 98
[Conditional] **trace** menu item 92
condition-bind special form 133
condition flavor 133
Conditions 201
Signalling conditions 133
Signalling Conditions: Program Development Tools and Techniques 133
:cond option for **trace** 92
:cond option to **trace** 276
:cond trace Option 276
Controlling Compiler Warnings 323
Controlling the Evaluation of Top-level Forms 314
Controlling the Format of **trace** Output 279
Manipulating the control stack 264, 271
Converting Lgp to Screen Coordinates 121
Converting Lgp to Screen Coordinates 121
Copying buffers 63
Copying Buffers and Files: Program Development Tools and Techniques 63
Copying files 63
c-P Debugger command 80, 264, 271
c-R Debugger command 80, 264, 271
Creating a File: Program Development Tools and Techniques 9
Creating a logical host 217
Creating files 9
Reverse Search (c-R) Zmacs command 57
c-S Debugger command 264, 271
Quick Arglist (c-sh-A) Zmacs command 43
Compile Region (c-sh-C) Zmacs command 70
c-sh-C Zmacs command 316
Brief Documentation (c-sh-D) Zmacs command 39, 42
Evaluate Region (c-sh-E) Zmacs command 75
Macro Expand Expression (c-sh-M) Zmacs command 100
c-sh-P Debugger command 264, 271
Describe Variable At Point (c-sh-V) Zmacs command 39
Set Pop Mark (c-SPACE) Zmacs command 59
Incremental Search (c-S) Zmacs command 57
c-T Debugger command 264, 271
Clearing the Trap-on-exit Flag for the Current and All Outer Frames 266
Setting the Trap-on-exit Flag for the Current and All Outer Frames 266
Toggling the Trap-on-exit Flag for the Current Frame 266
Current patch 242
Current stack frame 264, 271
Debugger functions to return values in current stack frame 267
Examining the Current Stack Frame in the Debugger 262
Mouse cursor as an arrow 297
c-U Stepper command 94
One Window (c-X 1) Zmacs command 65
Two Windows (c-X 2) Zmacs command 65
View Two Windows (c-X 3) Zmacs command 65
Modified Two Windows (c-X 4) Zmacs command 65
Set Comment Column (c-X ;) Zmacs command 23
Select Buffer (c-X B) Zmacs command 59
Display Directory (c-X c-D) Zmacs command 45
Find File (c-X c-F) Zmacs command 9

Swap Point And Mark	(c-X c-X) Zmacs command 59
	c-X Debugger command 271
↑r Dired	(c-X D) Zmacs command 45
Call Last Kbd Macro	(c-X E) Zmacs command 64
Set Fill Column	(c-X F) Zmacs command 12
Open Get Register	(c-X G) Zmacs command 63
Mark Whole	(c-X H) Zmacs command 63
Jump To Saved Position	(c-X J) Zmacs command 59
Other Window	(c-X O) Zmacs command 65
	c-X Stepper command 94
Save Position	(c-X S) Zmacs command 59
Put Register	(c-X X) Zmacs command 63
Start Kbd Macro	c-X () Zmacs command 64
Swap Point And Mark (c-X	c-X) Zmacs command 59
End Kbd Macro	c-X)]) Zmacs command 64
Yank	(c-Y) Zmacs command 60
	c-Z Inspector command 300
	c-Z in the Debugger 264, 271
Replace	(c-%) Zmacs command 57
Quit	(c-Z) Zmacs command 80

D

	:daemon method combination 113, 116, 121
	Daemon methods 116
Compiler Warnings	Database 321
Print compiler warnings	database 321
Update compiler warnings	database 321
The Compiler Warnings	Database: Program Development Tools and Techniques 79
Putting	Data in Compiled Code Files 331
	dbg:arg function 267
	dbg:*debug-io-override* variable 268
	dbg:*defer-package-dwim* variable 268
	dbg:*frame* variable 268
	dbg function 98, 256
Entering the Debugger with the	**dbg** Function 256
	dbg:fun function 268
	dbg:loc function 267
	dbg:*show-backtrace* variable 268
	dbg:val function 268
	Debugger 79, 80, 98, 253
c-Z in the	Debugger 264, 271
Display	Debugger 80
Entering the	Debugger 255
Entering the Display Debugger From the	Debugger 265
Entering the Editor From the	Debugger 264
Evaluating a Form in the	Debugger 259
Examining the Current Stack Frame in the	Debugger 262
Functions used inside the	Debugger 267
How to Use the	Debugger 259
Sending a Bug Report in the	Debugger 264
Exiting From the	Debugger: Abort 261
Examining Stack Frames with	Debugger Backtrace Commands 262
Entering the	Debugger by Causing an Error 255
ABORT	Debugger command 264, 271
c-A	Debugger command 80, 271
c-B	Debugger command 264, 271
c-E	Debugger command 80, 264, 271
c-HELP	Debugger command 80, 264, 271
c-L	Debugger command 80, 264, 271
c-M	Debugger command 264, 271
c-m-A	Debugger command 264, 271

c-m-B Debugger command 264, 271
c-m-F Debugger command 264, 271
c-m-H Debugger command 264, 271
c-m-L Debugger command 264, 271
c-m-N Debugger command 264, 271
c-m-P Debugger command 264, 271
c-m-R Debugger command 80, 264, 271
c-m-S Debugger command 264, 271
c-m-U Debugger command 264, 271
c-m-V Debugger command 264, 271
c-m-W Debugger command 80, 264, 271
c-m-X Debugger command 264, 271
c-N Debugger command 80, 264, 271
c-P Debugger command 80, 264, 271
c-R Debugger command 80, 264, 271
c-S Debugger command 264, 271
c-sh-P Debugger command 264, 271
c-T Debugger command 264, 271
c-X Debugger command 271
HELP Debugger command 264, 271
m-< Debugger command 264, 271
m-> Debugger command 264, 271
m-B Debugger command 80, 264, 271
m-I Debugger command 264, 267, 271
m-L Debugger command 80, 264, 271
m-N Debugger command 264, 271
m-P Debugger command 264, 271
m-S Debugger command 264, 271
m-X Debugger command 264, 271
RESUME Debugger command 264, 271
s-sh-C Debugger command 264, 271
Debugger commands 271
Summary of Debugger Commands 271
Debugger Commands for Dynamic Breakpoints and Stepping Through Compiled Code 267
Debugger Commands for Information Display 265
Debugger Commands for Stack Manipulation 263
Debugger Commands That Call Other Systems 264
Debugger Commands That Trap on Frame Exit 266
Entering the Display Debugger From the Debugger 265
Debugger Functions 267
Debugger functions to return values in current stack frame 267
Debugger Help 261
[Edit] Display Debugger menu item 80
[Retry] Display Debugger menu item 80
Debugger Proceed and Restart Options 256
The Debugger: Program Development Tools and Techniques 80
Proceeding From the Error in the Debugger: Resume 261
Debugger special commands 80
Debugger Variables 268
Entering the Debugger with m-SUSPEND 256
Entering the Debugger with the **dbg** Function 256
Debugging 79
Summary of Debugging Aids 273
Debugging Lisp Programs 79
dbg: ***debug-io-override*** variable 268
[DeCache] Inspector menu item 297, 298
System Declaration File 218
declare 315
def 310
:default-init-plist option for **defflavor** 116

defconst 310
defconstant 310
defconst special form 70
dbg: *defer-package-dwim* variable 268
deff 310
defflavor 310
:default-init-plist option for **defflavor** 116
:documentation option for **defflavor** 141
:gettable-instance-variables option for **defflavor** 113
:initable-instance-variables option for **defflavor** 124, 126
:required-flavors option for **defflavor** 116
:required-methods option for **defflavor** 113
defflavor macro 113, 116
si: **define-defsystem-special-variable** special form 214
si: **define-make-system-special-variable** special form 227
si: **define-simple-transformation** special form 214
Defining a System 191
Loading the System Definition 217
Mark Definition (c-m-H) Zmacs command 60
Edit Definition (m-.) Zmacs command 40, 141, 142
Add Patch Changed Definitions (m-X) 241
Compile Changed Definitions (m-X) Zmacs command 70
Edit Changed Definitions (m-X) Zmacs command 56
Evaluate Changed Definitions (m-X) Zmacs command 75
List Changed Definitions (m-X) Zmacs command 56
Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 70
Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 75
Add Patch Changed Definitions of Buffer (m-X) 241
Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
List Changed Definitions Of Buffer (m-X) Zmacs command 56
Definitions of functions 329
Definitions: Program Development Tools and Techniques 40
Loading System Definitions That Use Logical Pathnames 217
Loading System Definitions That Use Physical Pathnames 219
defmacro 310
defmethod 310
defpackage 310
defselect 310
defstruct 310
defsubst 310
defsystem 310
Adding New Options to **defsystem** 214
:bug-reports Option for **defsystem** 196
:compile-load-init Transformation of **defsystem** 211
:compile-load Transformation of **defsystem** 210
:compile Transformation of **defsystem** 209
:component-systems Option for **defsystem** 197
:do-components Transformation of **defsystem** 210
:fasload Transformation of **defsystem** 209
:initial-status Option for **defsystem** 196
:load-bfd Transformation of **defsystem** 210
:maintaining-sites Option for **defsystem** 196
:module Option for **defsystem** 199
:name Option for **defsystem** 194
:not-in-disk-label Option for **defsystem** 196
:package Option for **defsystem** 194
:patchable Option for **defsystem** 195, 231, 235
patch-atom argument to **:patchable** option for **defsystem** 236
:pathname-default Option for **defsystem** 195, 231, 235
:readfile Transformation of **defsystem** 209
:short-name Option for **defsystem** 194

:skip	**defsystem** Macro 213
	defsystem Modules 198
	defsystem Options 194
	defsystem special form 57, 191, 231
	defsystem Transformations 201, 210, 211
List of	**defsystem** Transformations 209
Interaction Between	**defsystem** Transformations and **make-system** 202
	defun 310
	defvar 310
	defvar special form 16, 70
	defvar-standard 310
	defwindow-resource special form 126
	Deinstall Macro (m-X) Zmacs command 64
Program Development: Refining Stripe	Density and Spacing 45
	Dependencies 201
	Deriving Methods for Tools and Techniques 3
Features	Described in Tools and Techniques 4
	describe-flavor function 141
	Describe Flavor (m-X) Zmacs command 141
	describe function 35, 104
	describe-system function 249
	Describe Variable At Point (c-sh-V) Zmacs command 39
Program Development:	Design and Figure Outline 13
	Designing the Advice 283
Program	Development: Design and Figure Outline 13
Program	Development: Drawing Stripes 27
Program	Development: Modifying the Output Module 112
Program	Development: Refining Stripe Density and Spacing 45
Aligning Code: Program	Development Tools and Techniques 26
Argument Lists: Program	Development Tools and Techniques 43
Balancing Parentheses: Program	Development Tools and Techniques 26
Before You Begin: Program	Development Tools and Techniques 7
Breakpoints: Program	Development Tools and Techniques 98
Callers: Program	Development Tools and Techniques 44
Commenting Out Code: Program	Development Tools and Techniques 83
Comments: Program	Development Tools and Techniques 23
Completion: Program	Development Tools and Techniques 8
Copying Buffers and Files: Program	Development Tools and Techniques 63
Creating a File: Program	Development Tools and Techniques 9
Definitions: Program	Development Tools and Techniques 40
Documentation: Program	Development Tools and Techniques 42
Editing Code: Program	Development Tools and Techniques 56
Entering Zmacs: Program	Development Tools and Techniques 9
Expanding Macros: Program	Development Tools and Techniques 100
File Attribute Lists: Program	Development Tools and Techniques 10
Functions: Program	Development Tools and Techniques 40
General Information on Flavors: Program	Development Tools and Techniques 141
Getting Started: Program	Development Tools and Techniques 9
HELP: Program	Development Tools and Techniques 7
Identifying Changed Code: Program	Development Tools and Techniques 56
Init Keywords: Program	Development Tools and Techniques 144
Keyboard Macros: Program	Development Tools and Techniques 64
Killing and Yanking: Program	Development Tools and Techniques 60
Lisp Input Editing: Program	Development Tools and Techniques 77
Major and Minor Modes: Program	Development Tools and Techniques 12
Methods: Program	Development Tools and Techniques 142
Moving Text: Program	Development Tools and Techniques 59
Moving Through Text: Program	Development Tools and Techniques 59
Multiple Buffers: Program	Development Tools and Techniques 65
Names: Program	Development Tools and Techniques 41
Objects: Program	Development Tools and Techniques 35

Other Displays: Program Development Tools and Techniques 68
Outlining the Figure: Program Development Tools and Techniques 16
Pathnames: Program Development Tools and Techniques 45
Program Development Tools and Techniques 1
Program Strategy: Program Development Tools and Techniques 13
Searching and Replacing: Program Development Tools and Techniques 57
Signalling Conditions: Program Development Tools and Techniques 133
Simple Screen Output: Program Development Tools and Techniques 14
Stepping: Program Development Tools and Techniques 94
Symbols: Program Development Tools and Techniques 38
The Compiler Warnings Database: Program Development Tools and Techniques 79
The Debugger: Program Development Tools and Techniques 80
The Inspector: Program Development Tools and Techniques 104
Tracing and Stepping: Program Development Tools and Techniques 92
Tracing: Program Development Tools and Techniques 92
Using Multiple Windows: Program Development Tools and Techniques 65
Using Registers: Program Development Tools and Techniques 63
Variables: Program Development Tools and Techniques 39
Zmacs and Other Windows: Program Development Tools and Techniques 66
Compiler Tools and Their Differences 316
Directories 45
Display Directory (c-X c-D) Zmacs command 45
File types of the patch directory file 236
Patch Directory File 234
View Directory (m-X) Zmacs command 45
↑r Dired (c-X D) Zmacs command 45
Dired (m-X) Zmacs command 45
disassemble function 104
Disassemble (m-X) Zmacs command 104
Debugger Commands for Information Display 265
Display status of file system display 303
Error Display 255
Inspection Pane Display 299
Display Debugger 80
Entering the Display Debugger From the Debugger 265
[Edit] Display Debugger menu item 80
[Retry] Display Debugger menu item 80
Display Directory (c-X c-D) Zmacs command 45
Other Displays: Program Development Tools and Techniques 68
Display status of active processes 303
Display status of areas 303
Display status of file system display 303
Display status of hostat 303
Display status of window area 303
Display system information 304
:do-components Transformation of **defsystem** 210
Brief Documentation (c-sh-D) Zmacs command 39, 42
documentation function 39, 42
Long Documentation (m-sh-D) Zmacs command 39, 42
:documentation option for **defflavor** 141
Documentation: Program Development Tools and Techniques 42
Mouse documentation string 129
Documentation strings 39, 42
Down Comment Line (m-N) Zmacs command 23
Program Development: Drawing Stripes 27
:draw-line method of **tv:graphics-mixin** 14, 116
bin file dumper 309, 316
sys: **dump-forms-to-file** function 331
Rebound Variable Bindings During Evaluation 260
Debugger Commands for Dynamic Breakpoints and Stepping Through Compiled Code 267

↑r Dired (c-X D) Zmacs command 45

E

SELECT E 9
:**edges-from** init option for **tv:essential-window** 116
Edit Callers (m-X) Zmacs command 44
Multiple Edit Callers (m-X) Zmacs command 44
Edit Changed Definitions (m-X) Zmacs command 56
Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
Edit Combined Methods (m-X) Zmacs command 142
Edit Compiler Warnings (m-X) Zmacs command 79, 321
Edit Definition (m-.) Zmacs command 40, 141, 142
[Edit] Display Debugger menu item 80
Edit File Warnings (m-X) Zmacs command 321
Lisp Input editing 77
Writing and Editing Code 7
Editing Code: Program Development Tools and Techniques 56
Lisp Input Editing: Program Development Tools and Techniques 77
Edit Methods (m-X) Zmacs command 65, 142
Evaluation and the Editor 75
c-m-Y input editor command 77
m-Y input editor command 77
Entering the Editor From the Debugger 264
Tools for Compiling Code From the Editor Into Your World 316
[Edit Screen] System menu item 66, 129
[Edit] System menu item 9
Electric Shift Lock Mode (m-X) Zmacs command 12
End Kbd Macro (c-X)) Zmacs command 64
END Zmacs minibuffer command 8
Entering and Leaving the Inspector 295
Entering the Debugger 255
Entering the Debugger by Causing an Error 255
Entering the Debugger with m-SUSPEND 256
Entering the Debugger with the **dbg** Function 256
Entering the Display Debugger From the Debugger 265
Entering the Editor From the Debugger 264
Entering Zmacs: Program Development Tools and Techniques 9
:entrycond option for **trace** 92
:entrycond option to **trace** 276
:entrycond trace Option 276
:entry option for **trace** 92
:entry option to **trace** 277
:entryprint option for **trace** 92
:entryprint option to **trace** 277
:entryprint trace Option 277
:entry trace Option 277
Specifying compiler environments 329
Entering the Debugger by Causing an Error 255
Error Display 255
error flavor 133
Proceeding From the Error in the Debugger: Resume 261
:error option for **trace** 92, 98
:error option to **trace** 276
error-restart-loop special form 129
[Error] **trace** menu item 92, 98
:error trace Option 276

ESCAPE Inspector command 300
:edges-from init option for **tv:** **essential-window** 116
:expose-p init option for **tv:** **essential-window** 116
:minimum-height init option for **tv:** **essential-window** 116
:minimum-width init option for **tv:** **essential-window** 116
evalhook 289
evalhook function 289
evalhook variable 289
Evaluate And Replace Into Buffer (m-X) Zmacs command 75
Evaluate Buffer (m-X) Zmacs command 75
Evaluate Changed Definitions (m-X) Zmacs command 75
Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 75
Evaluate Into Buffer (m-X) Zmacs command 75
Evaluate Minibuffer (m-ESCAPE) Zmacs command 75
Evaluate Region (c-sh-E) Zmacs command 75
Evaluating a Form in the Debugger 259
Evaluating code 69, 75, 94
Compiling and Evaluating Lisp 69
Evaluating Lisp Code 75
Rebound Variable Bindings During Evaluation 260
Stepping Through an Evaluation 287
Evaluation and the Editor 75
Controlling the Evaluation of Top-level Forms 314
eval-when 310
eval-when special form 314
Examining a Compiled Code File 300
Examining Stack Frames with Debugger Backtrace Commands 262
Examining the Current Stack Frame in the Debugger 262
Examining values of instance variables 267
Tracing Function Execution 275
Untracing Function Execution 280
Finding Out About Existing Code 35
Debugger Commands That Trap on Frame Exit 266
:exitbreak option for **trace** 92, 98
:exitbreak option to **trace** 276
:exitbreak trace Option 276
:exitcond option for **trace** 92
:exitcond option to **trace** 276
:exitcond trace Option 276
Exiting From the Debugger: Abort 261
Exiting the Inspector 295
[Exit] Inspector menu item 104, 298
:exit option for **trace** 92
:exit option to **trace** 278
:exitprint option for **trace** 92
:exitprint option to **trace** 277
:exitprint trace Option 277
:exit trace Option 277
Macro Expand Expression All (m-X) Zmacs command 100
Macro Expand Expression (c-sh-M) Zmacs command 100
Expanding macros 100
Expanding Macros: Program Development Tools and Techniques 100
:experimental system status 251
Push Pop Point Explicit (m-SPACE) Zmacs command 59
:expose-p init option for **tv:essential-window** 116
Macro Expand Expression All (m-X) Zmacs command 100
Macro Expand Expression (c-sh-M) Zmacs command 100

	***expr** special form 325
Call Last Kbd Macro (c-X	E) Zmacs command 64

F

Introduction to the System	Facility 189
Patch	Facility 231
System	facility 189
	:fasd-form message 331
	Fasdump 331
	:fasload simple transformation 201
	:fasload Transformation of **defsystem** 209
Optimizer	feature of the compiler 327
	Features Described in Tools and Techniques 4
	***fexpr** special form 325
A Mixin to Position the	Figure 113
Program Development: Design and	Figure Outline 13
Outlining the	Figure: Program Development Tools and Techniques 16
Add region to patch	file 238
Compiling and Loading a	File 73
Examining a Compiled Code	File 300
File types of the patch directory	file 236
File types of the system version-directory	file 236
Install patch	file 242
Patch	file 231
Patch Directory	File 234
Sys:site;*Logical-host*.Translations	File 218
Sys:site;*System-name*.System	File 217
System	file 217
System Declaration	File 218
System Version-directory	File 233
Translations	file 218
Sys:site;*system-name*.system	file 218
	File Attribute Lists: Program Development Tools and Techniques 10
Find	File (c-X c-F) Zmacs command 9
compiler:	**file-declaration** function 325
compiler:	**file-declare** function 325
bin	file dumper 309, 316
sys:	**file-local-declarations** variable 324
Compile	File (m-X) Zmacs command 73
Insert	File (m-X) Zmacs command 63
Load	File (m-X) Zmacs command 73
Set Patch	File (m-X) Zmacs command 10
Format of patch	file names 236
Creating a	File: Program Development Tools and Techniques 9
Arrays in compiled code	files 331
Compiled code objects in compiled code	files 331
Copying	files 63
Creating	files 9
File Types of Lisp Source and Compiled Code	Files 318
Individual Patch	Files 235
Init	files 12, 64
Instances in compiled code	files 331
Lists in compiled code	files 331
Names of Patch	Files 236
Numbers in compiled code	files 331
Organization of Patch	Files 235
Putting Data in Compiled Code	Files 331
Symbols in compiled code	files 331
System source	files 231
Tools for Compiling	Files 317

Types of Patch Files 233
Attribute lists (in files) 10
Copying Buffers and Files: Program Development Tools and Techniques 63
Files That Maclisp Must Compile 329
Display status of file system display 303
si: ***file-transformation-function*** variable 229
bin file type 318
File Types of Lisp Source and Compiled Code Files 318
File types of the patch directory file 236
File types of the system version-directory file 236
Edit File Warnings (m-X) Zmacs command 321
Set Fill Column (c-X F) Zmacs command 12
Fill Long Comment (m-X) Zmacs command 23
Auto Fill Mode (m-X) Zmacs command 12
Find File (c-X c-F) Zmacs command 9
Finding Out About Existing Code 35
Find Unbalanced Parentheses (m-X) Zmacs command 26
Finish Patch (m-X) 242
Clearing the Trap-on-exit Flag for the Current and All Outer Frames 266
Setting the Trap-on-exit Flag for the Current and All Outer Frames 266
Toggling the Trap-on-exit Flag for the Current Frame 266
condition flavor 133
error flavor 133
lgp:basic-lgp-stream flavor 124
si:vanilla-flavor flavor 142
sys:abort flavor 129
tv:any-tyi-mixin flavor 129
tv:graphics-mixin flavor 116
tv:list-mouse-buttons-mixin flavor 129
tv:process-mixin flavor 129
tv:sheet flavor 116, 133
tv:window flavor 112, 116
si: **flavor-allowed-init-keywords** function 144
Describe Flavor (m-X) Zmacs command 141
Init keywords (for flavors) 144
Programming Aids for Flavors and Windows 141
Using Flavors and Windows 111
Flavors for Lgp Output 124
General Information on Flavors: Program Development Tools and Techniques 141
Set Fonts (m-X) Zmacs command 10
Init keywords (for flavors) 144
advise special form 281
advise-within special form 285
break special form 98
compiler:add-optimizer special form 327
compiler:make-message-obsolete special form 324
compiler:make-obsolete special form 323
condition-bind special form 133
defconst special form 70
defsystem special form 57, 191, 231
defvar special form 16, 70
defwindow-resource special form 126
error-restart-loop special form 129
eval-when special form 314
***expr** special form 325
***fexpr** special form 325
***lexpr** special form 325
multiple-value special form 116
si:define-defsystem-special-variable special form 214

si:define-make-system-special-variable special form 227
si:define-simple-transformation special form 214
special special form 314
trace special form 92, 94, 98, 275
unadvise special form 282
unadvise-within special form 285
unspecial special form 314
untrace special form 92, 280
unwind-protect special form 133
with-open-stream special form 126
Format of patch file names 236
Controlling the Format of **trace** Output 279
Evaluating a Form in the Debugger 259
Controlling the Evaluation of Top-level Forms 314
How the Stream Compiler Handles Top-level Forms 310
Current stack frame 264, 271
Debugger functions to return values in current stack frame 267
Inspecting a stack frame 299
Toggling the Trap-on-exit Flag for the Current Frame 266
Debugger Commands That Trap on Frame Exit 266
Examining the Current Stack Frame in the Debugger 262
Clearing the Trap-on-exit Flag for the Current and All Outer Frames 266
Setting the Trap-on-exit Flag for the Current and All Outer Frames 266
Examining Stack Frames with Debugger Backtrace Commands 262
dbg: ***frame*** variable 268
Entering the Display Debugger From the Debugger 265
Entering the Editor From the Debugger 264
Exiting From the Debugger: Abort 261
Tools for Compiling Code From the Editor Into Your World 316
Proceeding From the Error in the Debugger: Resume 261
fs:make-logical-pathname-host 217
fs:set-logical-pathname-host 218
Advising a Function 281
applyhook function 291
apropos function 38
arglist function 43
breakon function 98
bug function 264
compile function 307, 319
compiler:compile-file function 73, 307, 317
compiler:compile-file-load function 73, 318
compiler:file-declaration function 325
compiler:file-declare function 325
compiler:function-defined function 325
compiler:function-referenced function 325
compiler:load-compiler-warnings function 321
dbg function 98, 256
dbg:arg function 267
dbg:fun function 268
dbg:loc function 267
dbg:val function 268
describe function 35, 104
describe-flavor function 141
describe-system function 249
disassemble function 104
documentation function 39, 42
Entering the Debugger with the **dbg** Function 256
evalhook function 289
get-handler-for function 142
inspect function 104, 295
listarray function 35

load function 73
load-and-save-patches function 246
load-patches function 245
make-system function 79, 217, 221, 307
mexp function 100
note-private-patch function 244
peek function 304
pkg-goto function 16
plist function 38
print-compiler-warnings function 321
print-system-modifications function 250
prompt-and-read function 133
set-system-status function 251
si:advise-1 function 283
si:flavor-allowed-init-keywords function 144
si:get-release-version function 250
si:get-system-version function 249
signal function 112, 133
si:make-hardcopy-stream function 126
si:map-system-files function 252
si:patch-loaded-p function 250
si:patch-system-pathname function 237
si:set-system-file-properties function 252
si:system-version-info function 249
si:unadvise-1 function 283
si:unbin-file function 300
step function 94, 287
sys:dump-forms-to-file function 331
The Top-level Function 126
tv:choose-variable-values function 126, 133
tv:make-blinker function 133
tv:make-window function 112, 121, 126, 129
tv:sheet-following-blinker function 133
typep function 141
unbreakon function 98
uncompile function 319
what-files-call function 38
where-is function 38
who-calls function 38
Function Apropos (m-X) Zmacs command 41
Function compiler 309, 315
compiler: **function-defined** function 325
Tracing Function Execution 275
Untracing Function Execution 280
Compiled function object 309
:function option for **tv:choose-variable-values** 133
Function-referenced-but-never-defined Warnings 324
compiler: **function-referenced** function 325
Functions 40
Advice to functions 281
Compiled functions 69
Debugger Functions 267
Definitions of functions 329
Interpreted functions 69
Tools for Compiling Single Functions 318
compiler: **functions-defined** variable 324
Functions: Program Development Tools and Techniques 40
compiler: **functions-referenced** variable 325
Functions That Operate on a System 251
Debugger functions to return values in current stack frame 267
Functions used inside the Debugger 267
Advising One Function Within Another 285

dbg: **fun** function 268
Set Fill Column (c-X F) Zmacs command 12

G

General Information on Flavors: Program Development Tools and Techniques 141
Generic operations 112
get-handler-for function 142
Open Get Register (c-X G) Zmacs command 63
si: **get-release-version** function 250
si: **get-system-version** function 249
:gettable-instance-variables option for **defflavor** 113
Getting Information About a System 249
Getting Started: Program Development Tools and Techniques 9
Graphic Output of the Sample Program 185
:draw-line method of **tv:** **graphics-mixin** 14, 116
tv: **graphics-mixin** flavor 116
Open Get Register (c-X G) Zmacs command 63

H

Restart handlers 80
How the Stream Compiler Handles Top-level Forms 310
Debugger Help 261
HELP Debugger command 264, 271
Peek Help Message 303
HELP: Program Development Tools and Techniques 7
HELP Stepper command 94
HELP Zmacs command 7, 64
HELP Zmacs minibuffer command 8
The Inspector History Pane 297
Creating a logical host 217
Display status of hostat 303
How the Inspector Works 295
How the Stream Compiler Handles Top-level Forms 310
How to Invoke the Compiler 307
How to Use the Debugger 259
Mark Whole (c-X H) Zmacs command 63

I

SELECT I 104, 295
Identifying Changed Code: Program Development Tools and Techniques 56
if-for-lispm macro 329
if-for-maclisp-else-lispm macro 329
if-for-maclisp macro 329
if-in-lispm macro 329
if-in-maclisp macro 329
Ignored arguments 324
ignore variable 324
Inactive patches 242
Incremental Search (c-S) Zmacs command 57
:increment-patch Option for **make-system** 223
Indent For Comment (c-; or m-;) Zmacs command 23
Indent For Lisp (TAB or c-m-TAB) Zmacs

command 26
Indent New Comment Line (m-LINE) Zmacs command 23
Indent New Line (LINE) Zmacs command 26
Indent Region (c-m-\) Zmacs command 26
Indent Sexp (c-m-Q) Zmacs command 26
Individual Patch Files 235
Attribute lists (in files) 10
Backtrace information 268
Display system information 304
Getting Information About a System 249
Debugger Commands for Information Display 265
General Information on Flavors: Program Development Tools and Techniques 141
inhibit-style-warnings macro 323
:initable-instance-variables option for **defflavor** 124, 126
Init files 12, 64
Initial patch state 242
:initial-status Option for **defsystem** 196
Init keywords (for flavors) 144
Init Keywords: Program Development Tools and Techniques 144
:init method of **tv:sheet** 116
:edges-from init option for **tv:essential-window** 116
:expose-p init option for **tv:essential-window** 116
:minimum-height init option for **tv:essential-window** 116
:minimum-width init option for **tv:essential-window** 116
:process init option for **tv:process-mixin** 129
:blinker-p init option for **tv:sheet** 116
In-progress patch 242
In-progress patch state 242
Lisp input editing 77
Lisp Input Editing: Program Development Tools and Techniques 77
c-m-Y input editor command 77
m-Y input editor command 77
Insert Buffer (m-X) Zmacs command 63
Insert File (m-X) Zmacs command 63
:inside-size method of **tv:sheet** 116
Functions used inside the Debugger 267
Inspect command 295
inspect function 104, 295
Inspecting a closure 299
Inspecting a compiled code object 299
Inspecting a list 299
Inspecting a named structure 299
Inspecting an array 299
Inspecting an instance 299
Inspecting a select method 299
Inspecting a stack frame 299
Inspecting a symbol 299
Inspecting objects 298
[Inspect] in system menu 295
The Inspector Inspection Pane 298
Inspection Pane Display 299
Inspector 104, 141, 293
Entering and Leaving the Inspector 295
Exiting the Inspector 295
Special Characters Recognized by the Inspector 300
The Inspector 293
Using the Inspector 295
Using the mouse in the Inspector 297, 298

BREAK	Inspector command 300
c-Z	Inspector command 300
ESCAPE	Inspector command 300
	Inspector commands 298
The	Inspector History Pane 297
The	Inspector Inspection Pane 298
The	Inspector Interaction Pane 297
[Clear]	Inspector menu item 298
[DeCache]	Inspector menu item 297, 298
[Exit]	Inspector menu item 104, 298
[Modify]	Inspector menu item 104, 298
[Return]	Inspector menu item 104, 298
[Set \]	Inspector menu item 298
The	Inspector Menu Pane 298
The	Inspector: Program Development Tools and Techniques 104
How the	Inspector Works 295
	[Inspect] System menu item 104
	Install Macro (m-X) Zmacs command 64
	Install Mouse Macro (m-X) Zmacs command 64
	Install patch file 242
Inspecting an	instance 299
	Instances in compiled code files 331
	Instance variables 113, 124, 126
Examining values of	instance variables 267
The Arrow Window:	Interaction, Processes, and the Mouse 129
	Interaction Between **defsystem** Transformations and **make-system** 202
The Inspector	Interaction Pane 297
	Interpreted functions 69
	Introduction to the Compiler 307
	Introduction to the System Facility 189
	Introduction to Tools and Techniques 3
How to	Invoke the Compiler 307
Where	Is Symbol (m-X) Zmacs command 38

J

	Jump To Saved Position (c-X J) Zmacs command 59
Jump To Saved Position (c-X	J) Zmacs command 59

K

Call Last	Kbd Macro (c-X E) Zmacs command 64
Name Last	Kbd Macro (m-X) Zmacs command 64
	Keeping Track of Lisp Syntax 23
	Keyboard macros 64
	Keyboard Macros: Program Development Tools and Techniques 64
Set	Key (m-X) Zmacs command 64
make-system	Keywords 222
Init	keywords (for flavors) 144
Init	Keywords: Program Development Tools and Techniques 144
Adding New	Keywords to **make-system** 227
	Kill Comment (c-m-;) Zmacs command 23
	Killing and Yanking: Program Development Tools and Techniques 60
	Killing text 60
	Kill Sexp (c-m-K) Zmacs command 60
Backward	Kill Sexp (c-m-RUBOUT) Zmacs command 60

L

Maintaining	Large Programs 187
Call	Last Kbd Macro (c-X E) Zmacs command 64
Name	Last Kbd Macro (m-X) Zmacs command 64
	:latest symbol in **make-system :version** option 222
Entering and	Leaving the Inspector 295
Patch	level 250
	***lexpr** special form 325
:send-command method of	**lgp:basic-lgp-stream** 124
:send-coordinates method of	**lgp:basic-lgp-stream** 124
	lgp:basic-lgp-stream flavor 124
Flavors for	Lgp Output 124
Converting	Lgp to Screen Coordinates 121
Indent New	Line (LINE) Zmacs command 26
Indent New Comment	Line (m-LINE) Zmacs command 23
Down Comment	Line (m-N) Zmacs command 23
Up Comment	Line (m-P) Zmacs command 23
	Line region 297
List Matching	Lines (m-X) Zmacs command 57
Indent New Line	(LINE) Zmacs command 26
Compiling and Evaluating	Lisp 69
Compiling	Lisp Code 70
Evaluating	Lisp Code 75
	Lisp input editing 77
	Lisp Input Editing: Program Development Tools and Techniques 77
	Lisp Mode (m-X) Zmacs command 12
Debugging	Lisp Programs 79
File Types of	Lisp Source and Compiled Code Files 318
Keeping Track of	Lisp Syntax 23
Indent For	Lisp (TAB or c-m-TAB) Zmacs command 26
Inspecting a	list 299
	listarray function 35
	List Callers (m-X) Zmacs command 38, 44
Multiple	List Callers (m-X) Zmacs command 44
	List Changed Definitions (m-X) Zmacs command 56
	List Changed Definitions Of Buffer (m-X) Zmacs command 56
	List Combined Methods (m-X) Zmacs command 142
	List Matching Lines (m-X) Zmacs command 57
	List Matching Symbols (m-X) Zmacs command 38
	List Methods (m-X) Zmacs command 142
tv:	**list-mouse-buttons-mixin** flavor 129
Reparse Attribute	List (m-X) Zmacs command 10
Update Attribute	List (m-X) Zmacs command 10
	List of **defsystem** Transformations 209
:system	*List* Option for **load-patches** 245
Argument	lists 43
	Lists in compiled code files 331
Attribute	lists (in files) 10
Argument	Lists: Program Development Tools and Techniques 43
File Attribute	Lists: Program Development Tools and Techniques 10
	load-and-save-patches function 246
	:load-bfd Transformation of **defsystem** 210
compiler:	**load-compiler-warnings** function 321
	Load Compiler Warnings (m-X) Zmacs command 79, 321
	Load File (m-X) Zmacs command 73
	load function 73
Compiling and	Loading a File 73

Loading patches 231, 244
Loading System Definitions That Use Logical Pathnames 217
Loading System Definitions That Use Physical Pathnames 219
Loading the System Definition 217
Load-like transformations 214
:noselective Option for **load-patches** 245
:nowarn Option for **load-patches** 246
:selective Option for **load-patches** 245, 246
:silent Option for **load-patches** 246
:system *List* Option for **load-patches** 245
System-name Option for **load-patches** 245
:verbose Option for **load-patches** 245, 246
load-patches function 245
Load System command 217
dbg: **loc** function 267
Electric Shift Lock Mode (m-X) Zmacs command 12
Creating a logical host 217
Loading System Definitions That Use Logical Pathnames 217
Fill Long Comment (m-X) Zmacs command 23
Long Documentation (m-sh-D) Zmacs command 39, 42
Set Lowercase (m-X) Zmacs command 10

M

Edit Definition (m-.) Zmacs command 40, 141, 142
Indent For Comment (c-; or m-;) Zmacs command 23
m-< Debugger command 264, 271
m-> Debugger command 264, 271
Files That Maclisp Must Compile 329
macro 310
#M sharp-sign reader macro 329
compile-flavor-methods macro 133
defflavor macro 113, 116
if-for-lispm macro 329
if-for-maclisp macro 329
if-for-maclisp-else-lispm macro 329
if-in-lispm macro 329
if-in-maclisp macro 329
inhibit-style-warnings macro 323
:skip defsystem Macro 213
#Q sharp-sign reader macro 329
Call Last Kbd Macro (c-X E) Zmacs command 64
Macro Expand Expression All (m-X) Zmacs command 100
Macro Expand Expression (c-sh-M) Zmacs command 100
Deinstall Macro (m-X) Zmacs command 64
Install Macro (m-X) Zmacs command 64
Install Mouse Macro (m-X) Zmacs command 64
Name Last Kbd Macro (m-X) Zmacs command 64
Expanding macros 100
Keyboard macros 64
Expanding Macros: Program Development Tools and Techniques 100
Keyboard Macros: Program Development Tools and Techniques 64
Bug mail 264, 271
Send mail about patch 242
Maintaining Large Programs 187
:maintaining-sites Option for **defsystem** 196

System maintenance 231
Major and Minor Modes: Program Development Tools and Techniques 12
Major version 236
Major version number 231
tv: **make-blinker** function 133
si: **make-hardcopy-stream** function 126
fs: **make-logical-pathname-host** 217
compiler: **make-message-obsolete** special form 324
compiler: **make-obsolete** special form 323
Adding New Keywords to **make-system** 227
:batch option for **make-system** 79, 222
:compile Option for **make-system** 222
:increment-patch Option for **make-system** 223
Interaction Between **defsystem** Transformations and **make-system** 202
:noconfirm Option for **make-system** 222
:no-increment-patch Option for **make-system** 223
:noload Option for **make-system** 223
:noop Option for **make-system** 223
:nowarn Option for **make-system** 222
:print-only Option for **make-system** 223
:recompile Option for **make-system** 223
:reload Option for **make-system** 223
:selective Option for **make-system** 222
:silent Option for **make-system** 222
:update-directory Option for **make-system** 226
:version Option for **make-system** 223
si: ***make-system-forms-to-be-evaled-after*** variable 229
si: ***make-system-forms-to-be-evaled-before*** variable 228
si: ***make-system-forms-to-be-evaled-finally*** variable 229
make-system function 79, 217, 221, 307
make-system Keywords 222
:latest symbol in **make-system :version** option 222
:newest symbol in **make-system :version** option 222
:released symbol in **make-system :version** option 222
tv: **make-window** function 112, 121, 126, 129
Making a System 221
Making Patches 238
Manipulating the control stack 264, 271
Debugger Commands for Stack Manipulation 263
si: **map-system-files** function 252
Set Pop Mark (c-SPACE) Zmacs command 59
Swap Point And Mark (c-X c-X) Zmacs command 59
Mark Definition (c-m-H) Zmacs command 60
Mark Whole (c-X H) Zmacs command 63
List Matching Lines (m-X) Zmacs command 57
List Matching Symbols (m-X) Zmacs command 38
m-B Debugger command 80, 264, 271
[Inspect] in system menu 295
[ARGPDL] **trace** menu item 92
[Attributes] System menu item 141
[Break after] **trace** menu item 92
[Break before] **trace** menu item 92
[Clear] Inspector menu item 298
[Cond after] **trace** menu item 92
[Cond before] **trace** menu item 92
[Cond break after] **trace** menu item 92, 98
[Cond break before] **trace** menu item 92, 98
[Conditional] **trace** menu item 92
[DeCache] Inspector menu item 297, 298

[Edit] Display Debugger menu item 80
[Edit Screen] System menu item 66, 129
[Edit] System menu item 9
[Error] **trace** menu item 92, 98
[Exit] Inspector menu item 104, 298
[Inspect] System menu item 104
[Modify] Inspector menu item 104, 298
[Per Process] **trace** menu item 92
[Print after] **trace** menu item 92
[Print before] **trace** menu item 92
[Print] **trace** menu item 92
[Retry] Display Debugger menu item 80
[Return] Inspector menu item 104, 298
[Set \] Inspector menu item 298
[Split Screen] System menu item 66, 129
[Step] **trace** menu item 92, 94
[Trace] System menu item 92, 94
[Untrace] **trace** menu item 92
[Wherein] **trace** menu item 92
The Inspector Menu Pane 298
Source Compare Merge (m-X) Zmacs command 56
Evaluate Minibuffer (m-ESCAPE) Zmacs command 75
:fasd-form message 331
Peek Help Message 303
Inspecting a select method 299
:proceed method 133
:report method 133
:who-line-documentation-string method 129
:case method combination 133
:daemon method combination 113, 116, 121
:send-command method of **lgp:basic-lgp-stream** 124
:send-coordinates method of **lgp:basic-lgp-stream** 124
:operation-handled-p method of **si:vanilla-flavor** 142
:which-operations method of **si:vanilla-flavor** 142
:any-tyi method of **tv:any-tyi-mixin** 129
:draw-line method of **tv:graphics-mixin** 14, 116
:change-of-size-or-margins method of **tv:sheet** 116
:init method of **tv:sheet** 116
:inside-size method of **tv:sheet** 116
:refresh method of **tv:sheet** 116, 129
Methods 142
Daemon methods 116
Primary methods 113, 116, 121
Deriving Methods for Tools and Techniques 3
Edit Methods (m-X) Zmacs command 65, 142
Edit Combined Methods (m-X) Zmacs command 142
List Methods (m-X) Zmacs command 142
List Combined Methods (m-X) Zmacs command 142
Methods: Program Development Tools and Techniques 142
mexp function 100
m-I Debugger command 264, 267, 271
Minibuffer 8
c-? Zmacs minibuffer command 8
COMPLETE Zmacs minibuffer command 8
END Zmacs minibuffer command 8
HELP Zmacs minibuffer command 8
RETURN Zmacs minibuffer command 8
SPACE Zmacs minibuffer command 8
Evaluate Minibuffer (m-ESCAPE) Zmacs command 75
:minimum-height init option for **tv:essential-window** 116
:minimum-width init option for

	tv:essential-window 116
Major and	Minor Modes: Program Development Tools and Techniques 12
	Minor version 236
	Minor version number 231, 238
A	Mixin to Position the Figure 113
	m-L Debugger command 80, 264, 271
Indent New Comment Line	(m-LINE) Zmacs command 23
	m-N Debugger command 264, 271
Down Comment Line	(m-N) Zmacs command 23
Atom Word	Mode (m-X) Zmacs command 12
Auto Fill	Mode (m-X) Zmacs command 12
Electric Shift Lock	Mode (m-X) Zmacs command 12
Lisp	Mode (m-X) Zmacs command 12
	Modes 12
Major and Minor	Modes: Program Development Tools and Techniques 12
Print	Modifications (m-X) Zmacs command 56
	Modified Two Windows (c-X 4) Zmacs command 65
Program Development:	Modifying the Output Module 112
	[Modify] Inspector menu item 104, 298
	Modularity 112
	Module 191
Anonymous	module 199
Program Development: Modifying the Output	Module 112
Calculation	Module for the Sample Program 147
Output	Module for the Sample Program 165
:package option for the	**:module** option 199
	:module Option for **defsystem** 199
defsystem	Modules 198
	Module specification 199
The Arrow Window: Interaction, Processes, and the	Mouse 129
	Mouse clicks 129
	Mouse cursor as an arrow 297
	Mouse documentation string 129
Using the	mouse in the Inspector 297, 298
Install	Mouse Macro (m-X) Zmacs command 64
	Move To Previous Point (c-m-SPACE) Zmacs command 59
	Moving text 59
	Moving Text: Program Development Tools and Techniques 59
	Moving Through Text: Program Development Tools and Techniques 59
	m-P Debugger command 264, 271
Up Comment Line	(m-P) Zmacs command 23
	m-S Debugger command 264, 271
Compile Changed Definitions Of Buffer	(m-sh-C) Zmacs command 70
Long Documentation	(m-sh-D) Zmacs command 39, 42
Evaluate Changed Definitions Of Buffer	(m-sh-E) Zmacs command 75
Push Pop Point Explicit	(m-SPACE) Zmacs command 59
Entering the Debugger with	m-SUSPEND 256
	Multiple buffers 65
	Multiple Buffers: Program Development Tools and Techniques 65
	Multiple Edit Callers (m-X) Zmacs command 44
	Multiple List Callers (m-X) Zmacs command 44
	multiple-value special form 116
	Multiple windows 65
Using	Multiple Windows: Program Development Tools and Techniques 65
Files That Maclisp	Must Compile 329
Save Region	(m-W) Zmacs command 60

Abort Patch (m-X) 243
Add Patch (m-X) 240
Add Patch Changed Definitions (m-X) 241
Add Patch Changed Definitions of Buffer (m-X) 241
Finish Patch (m-X) 242
Recompile Patch (m-X) 243
Reload Patch (m-X) 244
Resume Patch (m-X) 243
Select Patch (m-X) 242
Start Patch (m-X) 239
Start Private Patch (m-X) 240
View Patches (m-X) 242
m-X Debugger command 264, 271
Atom Word Mode (m-X) Zmacs command 12
Auto Fill Mode (m-X) Zmacs command 12
Compile Buffer (m-X) Zmacs command 70
Compile Changed Definitions (m-X) Zmacs command 70
Compile File (m-X) Zmacs command 73
Compile Region (m-X) Zmacs command 316
Compiler Warnings (m-X) Zmacs command 79, 321
Deinstall Macro (m-X) Zmacs command 64
Describe Flavor (m-X) Zmacs command 141
Dired (m-X) Zmacs command 45
Disassemble (m-X) Zmacs command 104
Edit Callers (m-X) Zmacs command 44
Edit Changed Definitions (m-X) Zmacs command 56
Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
Edit Combined Methods (m-X) Zmacs command 142
Edit Compiler Warnings (m-X) Zmacs command 79, 321
Edit File Warnings (m-X) Zmacs command 321
Edit Methods (m-X) Zmacs command 65, 142
Electric Shift Lock Mode (m-X) Zmacs command 12
Evaluate And Replace Into Buffer (m-X) Zmacs command 75
Evaluate Buffer (m-X) Zmacs command 75
Evaluate Changed Definitions (m-X) Zmacs command 75
Evaluate Into Buffer (m-X) Zmacs command 75
Fill Long Comment (m-X) Zmacs command 23
Find Unbalanced Parentheses (m-X) Zmacs command 26
Function Apropos (m-X) Zmacs command 41
Insert Buffer (m-X) Zmacs command 63
Insert File (m-X) Zmacs command 63
Install Macro (m-X) Zmacs command 64
Install Mouse Macro (m-X) Zmacs command 64
Lisp Mode (m-X) Zmacs command 12
List Callers (m-X) Zmacs command 38, 44
List Changed Definitions (m-X) Zmacs command 56
List Changed Definitions Of Buffer (m-X) Zmacs command 56
List Combined Methods (m-X) Zmacs command 142
List Matching Lines (m-X) Zmacs command 57
List Matching Symbols (m-X) Zmacs command 38
List Methods (m-X) Zmacs command 142
Load Compiler Warnings (m-X) Zmacs command 79, 321
Load File (m-X) Zmacs command 73
Macro Expand Expression All (m-X) Zmacs command 100
Multiple Edit Callers (m-X) Zmacs command 44
Multiple List Callers (m-X) Zmacs command 44
Name Last Kbd Macro (m-X) Zmacs command 64
Print Modifications (m-X) Zmacs command 56
Reparse Attribute List (m-X) Zmacs command 10
Select All Buffers As Tag Table (m-X) Zmacs command 57
Select System As Tag Table (m-X) Zmacs command 57
Set Backspace (m-X) Zmacs command 10
Set Base (m-X) Zmacs command 10

Set Fonts (m-X) Zmacs command 10
Set Key (m-X) Zmacs command 64
Set Lowercase (m-X) Zmacs command 10
Set Nofill (m-X) Zmacs command 10
Set Package (m-X) Zmacs command 10
Set Patch File (m-X) Zmacs command 10
Set Tab Width (m-X) Zmacs command 10
Set Vsp (m-X) Zmacs command 10
Source Compare (m-X) Zmacs command 56
Source Compare Merge (m-X) Zmacs command 56
Split Screen (m-X) Zmacs command 65
Tags Query Replace (m-X) Zmacs command 57
Tags Search (m-X) Zmacs command 57
Trace (m-X) Zmacs command 92, 94
Update Attribute List (m-X) Zmacs command 10
View Directory (m-X) Zmacs command 45
Where Is Symbol (m-X) Zmacs command 38
m-Y input editor command 77
Yank Pop (m-Y) Zmacs command 60
Query Replace (m-%) Zmacs command 57

N

Inspecting a named structure 299
Name Last Kbd Macro (m-X) Zmacs command 64
:name Option for **defsystem** 194
Format of patch file names 236
Names of Patch Files 236
Names: Program Development Tools and Techniques 41
Indent New Comment Line (m-LINE) Zmacs command 23
:newest symbol in **make-system :version** option 222
Adding New Keywords to **make-system** 227
Indent New Line (LINE) Zmacs command 26
Adding New Options to **defsystem** 214
:nil option for **trace** 92
:noconfirm Option for **make-system** 222
Set Nofill (m-X) Zmacs command 10
:no-increment-patch Option for **make-system** 223
:noload Option for **make-system** 223
:noop Option for **make-system** 223
:noselective Option for **load-patches** 245
note-private-patch function 244
:not-in-disk-label Option for **defsystem** 196
:nowarn Option for **load-patches** 246
:nowarn Option for **make-system** 222
Major version number 231
Minor version number 231, 238
Numbers in compiled code files 331

O

Compiled function object 309
Inspecting a compiled code object 299
Objects 35
Inspecting objects 298
Compiled code objects in compiled code files 331
Objects: Program Development Tools and Techniques 35
:obsolete system status 251
Attributes (of buffers) 10

Advising One Function Within Another 285
One Window (c-X 1) Zmacs command 65
Open Get Register (c-X G) Zmacs command 63
Functions That Operate on a System 251
:operation-handled-p method of **si:vanilla-flavor** 142
Generic operations 112
Optimizer feature of the compiler 327
Compiler Source-level Optimizers 327
:argpdl trace Option 277
:break trace Option 276
:cond trace Option 276
:entrycond trace Option 276
:entryprint trace Option 277
:entry trace Option 277
:error trace Option 276
:exitbreak trace Option 276
:exitcond trace Option 276
:exitprint trace Option 277
:exit trace Option 277
:latest symbol in **make-system :version** option 222
:newest symbol in **make-system :version** option 222
:per-process trace Option 277
:print trace Option 277
:released symbol in **make-system :version** option 222
:step trace Option 276
:wherein trace Option 277
:package option for the **:module** option 199
:default-init-plist option for **defflavor** 116
:documentation option for **defflavor** 141
:gettable-instance-variables option for **defflavor** 113
:initable-instance-variables option for **defflavor** 124, 126
:required-flavors option for **defflavor** 116
:required-methods option for **defflavor** 113
:bug-reports Option for **defsystem** 196
:component-systems Option for **defsystem** 197
:initial-status Option for **defsystem** 196
:maintaining-sites Option for **defsystem** 196
:module Option for **defsystem** 199
:name Option for **defsystem** 194
:not-in-disk-label Option for **defsystem** 196
:package Option for **defsystem** 194
:patchable Option for **defsystem** 195, 231, 235
patch-atom argument to **:patchable** option for **defsystem** 236
:pathname-default Option for **defsystem** 195, 231, 235
:short-name Option for **defsystem** 194
:noselective Option for **load-patches** 245
:nowarn Option for **load-patches** 246
:selective Option for **load-patches** 245, 246
:silent Option for **load-patches** 246
:system *List* Option for **load-patches** 245
System-name Option for **load-patches** 245
:verbose Option for **load-patches** 245, 246
:batch option for **make-system** 79, 222
:compile Option for **make-system** 222
:increment-patch Option for **make-system** 223
:noconfirm Option for **make-system** 222
:no-increment-patch Option for **make-system** 223
:noload Option for **make-system** 223
:noop Option for **make-system** 223
:nowarn Option for **make-system** 222
:print-only Option for **make-system** 223
:recompile Option for **make-system** 223

:reload Option for **make-system** 223
:selective Option for **make-system** 222
:silent Option for **make-system** 222
:update-directory Option for **make-system** 226
:version Option for **make-system** 223
:package option for the **:module** option 199
:arg option for **trace** 92
:argpdl option for **trace** 92
:both option for **trace** 92
:break option for **trace** 92, 98
:cond option for **trace** 92
:entry option for **trace** 92
:entrycond option for **trace** 92
:entryprint option for **trace** 92
:error option for **trace** 92, 98
:exit option for **trace** 92
:exitbreak option for **trace** 92, 98
:exitcond option for **trace** 92
:exitprint option for **trace** 92
:nil option for **trace** 92
:per-process option for **trace** 92
:print option for **trace** 92
:step option for **trace** 92, 94
:value option for **trace** 92
:wherein option for **trace** 92
:function option for **tv:choose-variable-values** 133
:edges-from init option for **tv:essential-window** 116
:expose-p init option for **tv:essential-window** 116
:minimum-height init option for **tv:essential-window** 116
:minimum-width init option for **tv:essential-window** 116
:process init option for **tv:process-mixin** 129
:blinker-p init option for **tv:sheet** 116
Debugger Proceed and Restart Options 256
defsystem Options 194
Adding New Options to **defsystem** 214
Options to **trace** 276
:arg option to **trace** 278
:argpdl option to **trace** 277
:both option to **trace** 278
:break option to **trace** 276
:cond option to **trace** 276
:entry option to **trace** 277
:entrycond option to **trace** 276
:entryprint option to **trace** 277
:error option to **trace** 276
:exit option to **trace** 278
:exitbreak option to **trace** 276
:exitcond option to **trace** 276
:exitprint option to **trace** 277
:print option to **trace** 277
:step option to **trace** 276
:value option to **trace** 278
:wherein option to **trace** 277
Organization of Patch Files 235
Organization of Tools and Techniques 4
Other Displays: Program Development Tools and Techniques 68
Debugger Commands That Call Other Systems 264
Scroll Other Window (c-m-V) Zmacs command 65
Other Window (c-X O) Zmacs command 65
Zmacs and Other Windows: Program Development Tools and Techniques 66
Finding Out About Existing Code 35

Commenting Out Code: Program Development Tools and Techniques 83
Clearing the Trap-on-exit Flag for the Current and All Outer Frames 266
Setting the Trap-on-exit Flag for the Current and All Outer Frames 266
Program Development: Design and Figure Outline 13
Outlining the Figure: Program Development Tools and Techniques 16
Controlling the Format of **trace** Output 279
Flavors for Lgp Output 124
trace output 279
Program Development: Modifying the Output Module 112
Output Module for the Sample Program 165
Graphic Output of the Sample Program 185
Simple Screen Output: Program Development Tools and Techniques 14
Overriding Variable-defined-but-never-referenced Warnings 326
Overview of Peek 303
Other Window (c-X O) Zmacs command 65

P P P

Set Package (m-X) Zmacs command 10
:package Option for **defsystem** 194
:package option for the **:module** option 199
Packages 10, 16, 38, 44
The Inspector History Pane 297
The Inspector Inspection Pane 298
The Inspector Interaction Pane 297
The Inspector Menu Pane 298
Inspection Pane Display 299
Balancing Parentheses 26
Find Unbalanced Parentheses (m-X) Zmacs command 26
Balancing Parentheses: Program Development Tools and Techniques 26
Patch 231
Current patch 242
In-progress patch 242
Send mail about patch 242
:patchable Option for **defsystem** 195, 231, 235
patch-atom argument to **:patchable** option for **defsystem** 236
Changing the Status of a Patchable System 251
patch-atom argument to **:patchable** option for **defsystem** 236
Add Patch Changed Definitions (m-X) 241
Add Patch Changed Definitions of Buffer (m-X) 241
Patch Directory File 234
File types of the patch directory file 236
Active patches 238, 242
Loading patches 231, 244
Making Patches 238
Inactive patches 242
View Patches (m-X) 242
Patch Facility 231
Patch file 231
Add region to patch file 238
Install patch file 242
Set Patch File (m-X) Zmacs command 10
Format of patch file names 236
Individual Patch Files 235
Names of Patch Files 236
Organization of Patch Files 235
Types of Patch Files 233

Patch level 250
si: **patch-loaded-p** function 250
Abort Patch (m-X) 243
Add Patch (m-X) 240
Finish Patch (m-X) 242
Recompile Patch (m-X) 243
Reload Patch (m-X) 244
Resume Patch (m-X) 243
Select Patch (m-X) 242
Start Patch (m-X) 239
Start Private Patch (m-X) 240
Initial patch state 242
In-progress patch state 242
si: **patch-system-pathname** function 237
:pathname-default Option for **defsystem** 195, 231, 235
Pathnames 45
Loading System Definitions That Use Logical Pathnames 217
Loading System Definitions That Use Physical Pathnames 219
Pathnames: Program Development Tools and Techniques 45
Overview of Peek 303
Set sleep time between updates Peek command 303
peek function 304
Peek Help Message 303
The Peek Program 301
:per-process option for **trace** 92
[Per Process] **trace** menu item 92
:per-process trace Option 277
Loading System Definitions That Use Physical Pathnames 219
pkg-goto function 16
plist function 38
Swap Point And Mark (c-X c-X) Zmacs command 59
Move To Previous Point (c-m-SPACE) Zmacs command 59
Describe Variable At Point (c-sh-V) Zmacs command 39
Push Pop Point Explicit (m-SPACE) Zmacs command 59
Set Pop Mark (c-SPACE) Zmacs command 59
Yank Pop (m-Y) Zmacs command 60
Push Pop Point Explicit (m-SPACE) Zmacs command 59
Jump To Saved Position (c-X J) Zmacs command 59
Save Position (c-X S) Zmacs command 59
A Mixin to Position the Figure 113
Prerequisites to Tools and Techniques 3
Select Previous Buffer (c-m-L) Zmacs command 59
Move To Previous Point (c-m-SPACE) Zmacs command 59
Primary methods 113, 116, 121
[Print after] **trace** menu item 92
[Print before] **trace** menu item 92
Print compiler warnings database 321
print-compiler-warnings function 321
Print Modifications (m-X) Zmacs command 56
:print-only Option for **make-system** 223
:print option for **trace** 92
:print option to **trace** 277
print-system-modifications function 250
[Print] **trace** menu item 92
:print trace Option 277
Start Private Patch (m-X) 240
Debugger Proceed and Restart Options 256
Proceeding 133
Proceeding From the Error in the Debugger: Resume 261
:proceed method 133

Proceed types 80, 133
Processes 129
Display status of active processes 303
The Arrow Window: Interaction, Processes, and the Mouse 129
:process init option for **tv:process-mixin** 129
:process init option for **tv:** **process-mixin** 129
tv: **process-mixin** flavor 129
[Per Process] **trace** menu item 92
progn 310
Calculation Module for the Sample Program 147
Graphic Output of the Sample Program 185
Output Module for the Sample Program 165
The Peek Program 301
Program Development: Design and Figure Outline 13
Program Development: Drawing Stripes 27
Program Development: Modifying the Output Module 112
Program Development: Refining Stripe Density and Spacing 45
Program Development Tools and Techniques 1
Aligning Code: Program Development Tools and Techniques 26
Argument Lists: Program Development Tools and Techniques 43
Balancing Parentheses: Program Development Tools and Techniques 26
Before You Begin: Program Development Tools and Techniques 7
Breakpoints: Program Development Tools and Techniques 98
Callers: Program Development Tools and Techniques 44
Commenting Out Code: Program Development Tools and Techniques 83
Comments: Program Development Tools and Techniques 23
Completion: Program Development Tools and Techniques 8
Copying Buffers and Files: Program Development Tools and Techniques 63
Creating a File: Program Development Tools and Techniques 9
Definitions: Program Development Tools and Techniques 40
Documentation: Program Development Tools and Techniques 42
Editing Code: Program Development Tools and Techniques 56
Entering Zmacs: Program Development Tools and Techniques 9
Expanding Macros: Program Development Tools and Techniques 100
File Attribute Lists: Program Development Tools and Techniques 10
Functions: Program Development Tools and Techniques 40
General Information on Flavors: Program Development Tools and Techniques 141
Getting Started: Program Development Tools and Techniques 9
HELP: Program Development Tools and Techniques 7
Identifying Changed Code: Program Development Tools and Techniques 56
Init Keywords: Program Development Tools and Techniques 144
Keyboard Macros: Program Development Tools and Techniques 64
Killing and Yanking: Program Development Tools and Techniques 60
Lisp Input Editing: Program Development Tools and Techniques 77
Major and Minor Modes: Program Development Tools and Techniques 12
Methods: Program Development Tools and Techniques 142
Moving Text: Program Development Tools and Techniques 59
Moving Through Text: Program Development Tools and Techniques 59
Multiple Buffers: Program Development Tools and Techniques 65
Names: Program Development Tools and Techniques 41
Objects: Program Development Tools and Techniques 35
Other Displays: Program Development Tools and Techniques 68
Outlining the Figure: Program Development Tools and Techniques 16
Pathnames: Program Development Tools and Techniques 45
Program Strategy: Program Development Tools and Techniques 13
Searching and Replacing: Program Development Tools and Techniques 57
Signalling Conditions: Program Development Tools and Techniques 133
Simple Screen Output: Program Development Tools and Techniques 14
Stepping: Program Development Tools and Techniques 94
Symbols: Program Development Tools and Techniques 38
The Compiler Warnings Database: Program Development Tools and Techniques 79

The Debugger: Program Development Tools and Techniques 80
The Inspector: Program Development Tools and Techniques 104
Tracing: Program Development Tools and Techniques 92
Tracing and Stepping: Program Development Tools and Techniques 92
Using Multiple Windows: Program Development Tools and Techniques 65
Using Registers: Program Development Tools and Techniques 63
Variables: Program Development Tools and Techniques 39
Zmacs and Other Windows: Program Development Tools and Techniques 66
Programming Aids for Flavors and Windows 141
Debugging Lisp Programs 79
Maintaining Large Programs 187
Program Strategy: Program Development Tools and Techniques 13
prompt-and-read function 133
compiler:top-level-form property 315
Purpose of Tools and Techniques 3
Push Pop Point Explicit (m-SPACE) Zmacs command 59
Put Register (c-X X) Zmacs command 63
Putting Data in Compiled Code Files 331

Q

query-io variable 133
Tags Query Replace (m-X) Zmacs command 57
Query Replace (m-%) Zmacs command 57
si: ***query-type*** variable 229
Quick Arglist (c-sh-A) Zmacs command 43
Quit (c-Z) Zmacs command 80

R

#M sharp-sign reader macro 329
#Q sharp-sign reader macro 329
:readfile Transformation of **defsystem** 209
Rebound Variable Bindings During Evaluation 260
Special Characters Recognized by the Inspector 300
:recompile Option for **make-system** 223
Recompile Patch (m-X) 243
si: ***redo-all*** variable 229
Program Development: Refining Stripe Density and Spacing 45
:refresh method of **tv:sheet** 116, 129
Compile Region 316
Line region 297
Indent Region (c-m-\) Zmacs command 26
Compile Region (c-sh-C) Zmacs command 70
Evaluate Region (c-sh-E) Zmacs command 75
Save Region (m-W) Zmacs command 60
Compile Region (m-X) Zmacs command 316
Add region to patch file 238
Open Get Register (c-X G) Zmacs command 63
Put Register (c-X X) Zmacs command 63
Registers 63
Using Registers: Program Development Tools and Techniques 63
:released symbol in **make-system :version** option 222
:released system status 251
:reload Option for **make-system** 223
Reload Patch (m-X) 244
Reparse Attribute List (m-X) Zmacs command 10
Replace (c-%) Zmacs command 57

Evaluate And Replace Into Buffer (m-X) Zmacs command 75
Tags Query Replace (m-X) Zmacs command 57
Query Replace (m-%) Zmacs command 57
Replacing 57
Searching and Replacing: Program Development Tools and Techniques 57
Sending a Bug Report in the Debugger 264
:report method 133
Bug reports 264, 271
:required-flavors option for **defflavor** 116
:required-methods option for **defflavor** 113
Resources 126
Restart handlers 80
Debugger Proceed and Restart Options 256
RESUME 75, 98
Proceeding From the Error in the Debugger: Resume 261
RESUME Debugger command 264, 271
Resume Patch (m-X) 243
[Retry] Display Debugger menu item 80
[Return] Inspector menu item 104, 298
Debugger functions to return values in current stack frame 267
RETURN Zmacs minibuffer command 8
Reverse Search (c-R) Zmacs command 57

S

Calculation Module for the Sample Program 147
Graphic Output of the Sample Program 185
Output Module for the Sample Program 165
Jump To Saved Position (c-X J) Zmacs command 59
Save Position (c-X S) Zmacs command 59
Save Region (m-W) Zmacs command 60
Scope of Tools and Techniques 3
Converting Lgp to Screen Coordinates 121
Split Screen (m-X) Zmacs command 65
Simple Screen Output: Program Development Tools and Techniques 14
[Edit Screen] System menu item 66, 129
[Split Screen] System menu item 66, 129
Scroll Other Window (c-m-V) Zmacs command 65
Reverse Search (c-R) Zmacs command 57
Incremental Search (c-S) Zmacs command 57
Searching 57
Searching and Replacing: Program Development Tools and Techniques 57
Tags Search (m-X) Zmacs command 57
Select Activity command 9
Select All Buffers As Tag Table (m-X) Zmacs command 57
Select Buffer (c-X B) Zmacs command 59
SELECT E 9
SELECT I 104, 295
:selective Option for **load-patches** 245, 246
:selective Option for **make-system** 222
Inspecting a select method 299
Select Patch (m-X) 242
Select Previous Buffer (c-m-L) Zmacs command 59
Select System As Tag Table (m-X) Zmacs command 57
:send-command method of **lgp:basic-lgp-stream** 124
:send-coordinates method of **lgp:basic-lgp-stream** 124

Sending a Bug Report in the Debugger 264
Send mail about patch 242
zwei: ***send-mail-about-patch*** 242
Set Backspace (m-X) Zmacs command 10
Set Base (m-X) Zmacs command 10
Set Comment Column (c-X ;) Zmacs command 23
Set Fill Column (c-X F) Zmacs command 12
Set Fonts (m-X) Zmacs command 10
Set Key (m-X) Zmacs command 64
fs: **set-logical-pathname-host** 218
Set Lowercase (m-X) Zmacs command 10
Set Nofill (m-X) Zmacs command 10
Set Package (m-X) Zmacs command 10
Set Patch File (m-X) Zmacs command 10
Set Pop Mark (c-SPACE) Zmacs command 59
Set sleep time between updates Peek command 303
si: **set-system-file-properties** function 252
si: **set-system-source-file** 217, 218, 219
set-system-status function 251
Set Tab Width (m-X) Zmacs command 10
Setting the Trap-on-exit Flag for the Current and All Outer Frames 266
Set Vsp (m-X) Zmacs command 10
[Set \] Inspector menu item 298
Kill Sexp (c-m-K) Zmacs command 60
Indent Sexp (c-m-Q) Zmacs command 26
Backward Kill Sexp (c-m-RUBOUT) Zmacs command 60
#M sharp-sign reader macro 329
#Q sharp-sign reader macro 329
:blinker-p init option for **tv: sheet** 116
:change-of-size-or-margins method of **tv: sheet** 116
:init method of **tv: sheet** 116
:inside-size method of **tv: sheet** 116
:refresh method of **tv: sheet** 116, 129
tv: sheet flavor 116, 133
tv: sheet-following-blinker function 133
Electric Shift Lock Mode (m-X) Zmacs command 12
:short-name Option for **defsystem** 194
dbg: *show-backtrace* variable 268
si:advise-1 function 283
si:advised-functions variable 283
si:*batch-mode-p* variable 229
si:define-defsystem-special-variable special form 214
si:define-make-system-special-variable special form 227
si:define-simple-transformation special form 214
si:*file-transformation-function* variable 229
si:flavor-allowed-init-keywords function 144
si:get-release-version function 250
si:get-system-version function 249
signal function 112, 133
Signalling conditions 133
Signalling Conditions: Program Development Tools and Techniques 133
:silent Option for **load-patches** 246
:silent Option for **make-system** 222
si: ***silent-p*** variable 229
si:make-hardcopy-stream function 126
si:*make-system-forms-to-be-evaled-after* variable 229
si:*make-system-forms-to-be-evaled-before* variable 228

si:*make-system-forms-to-be-evaled-finally* variable 229
si:map-system-files function 252
Simple Screen Output: Program Development Tools and Techniques 14
:fasload simple transformation 201
Tools for Compiling Single Functions 318
si:patch-loaded-p function 250
si:patch-system-pathname function 237
si:*query-type* variable 229
si:*redo-all* variable 229
si:set-system-file-properties function 252
si:set-system-source-file 217, 218, 219
si:*silent-p* variable 229
si:*system-being-defined* variable 214
si:*system-being-made* variable 228
si:system-version-info function 249
Sys: site;*Logical-host*.Translations File 218
Sys: site;*System-name*.System File 217
Sys: site;*system-name*.system file 218
si:*top-level-transformations* variable 229
si:*trace-bar-p* variable 279
si:*trace-bar-rate* variable 279
si:*trace-columns-per-level* variable 279
si:*trace-old-style* variable 280
si:unadvise-1 function 283
si:unbin-file function 300
:operation-handled-p method of **si:vanilla-flavor** 142
:which-operations method of **si:vanilla-flavor** 142
si:vanilla-flavor flavor 142
:skip defsystem Macro 213
Set sleep time between updates Peek command 303
File Types of Lisp Source and Compiled Code Files 318
Source Compare Merge (m-X) Zmacs command 56
Source Compare (m-X) Zmacs command 56
System source files 231
Compiler Source-level Optimizers 327
SPACE Stepper command 94
SPACE Zmacs minibuffer command 8
Program Development: Refining Stripe Density and Spacing 45
Special Characters Recognized by the Inspector 300
Debugger special commands 80
advise special form 281
advise-within special form 285
break special form 98
compiler:add-optimizer special form 327
compiler:make-message-obsolete special form 324
compiler:make-obsolete special form 323
condition-bind special form 133
defconst special form 70
defsystem special form 57, 191, 231
defvar special form 16, 70
defwindow-resource special form 126
error-restart-loop special form 129
eval-when special form 314
***expr** special form 325
***fexpr** special form 325
***lexpr** special form 325
multiple-value special form 116
si:define-defsystem-special-variable special form 214
si:define-make-system-special-variable special form 227
si:define-simple-transformation special form 214
special special form 314

trace	special form 92, 94, 98, 275
unadvise	special form 282
unadvise-within	special form 285
unspecial	special form 314
untrace	special form 92, 280
unwind-protect	special form 133
with-open-stream	special form 126
	special special form 314
Module	specification 199
	Specifying compiler environments 329
	Split Screen (m-X) Zmacs command 65
	[Split Screen] System menu item 66, 129
	s-sh-C Debugger command 264, 271
Backtrace of the call	stack 268
Manipulating the control	stack 264, 271
Current	stack frame 264, 271
Debugger functions to return values in current	stack frame 267
Inspecting a	stack frame 299
Examining the Current	Stack Frame in the Debugger 262
Examining	Stack Frames with Debugger Backtrace Commands 262
Debugger Commands for	Stack Manipulation 263
	standard-output variable 75
Getting	Started: Program Development Tools and Techniques 9
	Start Kbd Macro (c-X () Zmacs command 64
	Start Patch (m-X) 239
	Start Private Patch (m-X) 240
Initial patch	state 242
In-progress patch	state 242
:broken system	status 251
:experimental system	status 251
:obsolete system	status 251
:released system	status 251
System	status 303
Display	status of active processes 303
Changing the	Status of a Patchable System 251
Display	status of areas 303
Display	status of file system display 303
Display	status of hostat 303
Display	status of window area 303
	step function 94, 287
	:step option for **trace** 92, 94
	:step option to **trace** 276
	Stepper 94
c-B	Stepper command 94
c-E	Stepper command 94
c-N	Stepper command 94
c-U	Stepper command 94
c-X	Stepper command 94
HELP	Stepper command 94
SPACE	Stepper command 94
	Stepping 75, 94
	Stepping: Program Development Tools and Techniques 94
Tracing and	Stepping: Program Development Tools and Techniques 92
	Stepping Through an Evaluation 287
Debugger Commands for Dynamic Breakpoints and	Stepping Through Compiled Code 267
	[Step] **trace** menu item 92, 94
	:step trace Option 276
Program	Strategy: Program Development Tools and Techniques 13

Stream compiler 309
How the Stream Compiler Handles Top-level Forms 310
Mouse documentation string 129
Documentation strings 39, 42
Program Development: Refining Stripe Density and Spacing 45
Program Development: Drawing Stripes 27
Inspecting a named structure 299
Structure of the Compiler 309
Compiler Style Warnings 323
Summary of Compiler Actions on Code in a Zmacs Buffer 71
Summary of Debugger Commands 271
Summary of Debugging Aids 273
SUSPEND 75
Swap Point And Mark (c-X c-X) Zmacs command 59
Inspecting a symbol 299
:latest symbol in **make-system :version** option 222
:newest symbol in **make-system :version** option 222
:released symbol in **make-system :version** option 222
Where Is Symbol (m-X) Zmacs command 38
Symbols 38
Symbols in compiled code files 331
List Matching Symbols (m-X) Zmacs command 38
Symbols: Program Development Tools and Techniques 38
Keeping Track of Lisp Syntax 23
sys:abort flavor 129
sys:dump-forms-to-file function 331
sys:file-local-declarations variable 324
Sys:site;*Logical-host*.Translations File 218
Sys:site;*System-name*.System File 217
Sys:site;*system-name*.system file 218
System 189
Changing the Status of a Patchable System 251
Defining a System 191
Functions That Operate on a System 251
Getting Information About a System 249
Making a System 221
Select System As Tag Table (m-X) Zmacs command 57
si: *system-being-defined* variable 214
si: *system-being-made* variable 228
Compile System command 217
Load System command 217
System Declaration File 218
Loading the System Definition 217
Loading System Definitions That Use Logical Pathnames 217
Loading System Definitions That Use Physical Pathnames 219
Display status of file system display 303
System facility 189
Introduction to the System Facility 189
System file 217
Display system information 304
: system *List* Option for **load-patches** 245
System maintenance 231
[Inspect] in system menu 295
[Attributes] System menu item 141
[Edit] System menu item 9
[Edit Screen] System menu item 66, 129
[Inspect] System menu item 104
[Split Screen] System menu item 66, 129
[Trace] System menu item 92, 94
System-name Option for **load-patches** 245

Debugger Commands That Call Other Systems 264
Updating systems 231
System source files 231
System status 303
:broken system status 251
:experimental system status 251
:obsolete system status 251
:released system status 251
System Version-directory File 233
File types of the system version-directory file 236
si: **system-version-info** function 249
System versions 231
Save Position (c-X S) Zmacs command 59

T

Select All Buffers As Tag Table (m-X) Zmacs command 57
Select System As Tag Table (m-X) Zmacs command 57
Tag tables 57
Indent For Lisp (TAB or c-m-TAB) Zmacs command 26
Set Tab Width (m-X) Zmacs command 10
Tags Query Replace (m-X) Zmacs command 57
Tags Search (m-X) Zmacs command 57
Select All Buffers As Tag Table (m-X) Zmacs command 57
Select System As Tag Table (m-X) Zmacs command 57
Tag tables 57
Aligning Code: Program Development Tools and Techniques 26
Argument Lists: Program Development Tools and Techniques 43
Balancing Parentheses: Program Development Tools and Techniques 26
Before You Begin: Program Development Tools and Techniques 7
Breakpoints: Program Development Tools and Techniques 98
Callers: Program Development Tools and Techniques 44
Commenting Out Code: Program Development Tools and Techniques 83
Comments: Program Development Tools and Techniques 23
Completion: Program Development Tools and Techniques 8
Copying Buffers and Files: Program Development Tools and Techniques 63
Creating a File: Program Development Tools and Techniques 9
Definitions: Program Development Tools and Techniques 40
Deriving Methods for Tools and Techniques 3
Documentation: Program Development Tools and Techniques 42
Editing Code: Program Development Tools and Techniques 56
Entering Zmacs: Program Development Tools and Techniques 9
Expanding Macros: Program Development Tools and Techniques 100
Features Described in Tools and Techniques 4
File Attribute Lists: Program Development Tools and Techniques 10
Functions: Program Development Tools and Techniques 40
General Information on Flavors: Program Development Tools and Techniques 141
Getting Started: Program Development Tools and Techniques 9
HELP: Program Development Tools and Techniques 7
Identifying Changed Code: Program Development Tools and Techniques 56
Init Keywords: Program Development Tools and Techniques 144
Introduction to Tools and Techniques 3
Keyboard Macros: Program Development Tools and Techniques 64
Killing and Yanking: Program Development Tools and Techniques 60
Lisp Input Editing: Program Development Tools and Techniques 77
Major and Minor Modes: Program Development Tools and Techniques 12
Methods: Program Development Tools and Techniques 142

Moving Text: Program Development Tools and Techniques 59
Moving Through Text: Program Development Tools and Techniques 59
Multiple Buffers: Program Development Tools and Techniques 65
Names: Program Development Tools and Techniques 41
Objects: Program Development Tools and Techniques 35
Organization of Tools and Techniques 4
Other Displays: Program Development Tools and Techniques 68
Outlining the Figure: Program Development Tools and Techniques 16
Pathnames: Program Development Tools and Techniques 45
Prerequisites to Tools and Techniques 3
Program Development Tools and Techniques 1
Program Strategy: Program Development Tools and Techniques 13
Purpose of Tools and Techniques 3
Scope of Tools and Techniques 3
Searching and Replacing: Program Development Tools and Techniques 57
Signalling Conditions: Program Development Tools and Techniques 133
Simple Screen Output: Program Development Tools and Techniques 14
Stepping: Program Development Tools and Techniques 94
Symbols: Program Development Tools and Techniques 38
The Compiler Warnings Database: Program Development Tools and Techniques 79
The Debugger: Program Development Tools and Techniques 80
The Inspector: Program Development Tools and Techniques 104
Tracing and Stepping: Program Development Tools and Techniques 92
Tracing: Program Development Tools and Techniques 92
Using Multiple Windows: Program Development Tools and Techniques 65
Using Registers: Program Development Tools and Techniques 63
Variables: Program Development Tools and Techniques 39
Zmacs and Other Windows: Program Development Tools and Techniques 66
terminal-io variable 14, 112
Killing text 60
Moving text 59
Yanking text 60
Moving Text: Program Development Tools and Techniques 59
Moving Through Text: Program Development Tools and Techniques 59
Debugger Commands That Call Other Systems 264
Files That Maclisp Must Compile 329
Functions That Operate on a System 251
Debugger Commands That Trap on Frame Exit 266
Loading System Definitions That Use Logical Pathnames 217
Loading System Definitions That Use Physical Pathnames 219
Compiler Tools and Their Differences 316
Set sleep time between updates Peek command 303
Toggling the Trap-on-exit Flag for the Current Frame 266
Aligning Code: Program Development Tools and Techniques 26
Argument Lists: Program Development Tools and Techniques 43
Balancing Parentheses: Program Development Tools and Techniques 26
Before You Begin: Program Development Tools and Techniques 7
Breakpoints: Program Development Tools and Techniques 98
Callers: Program Development Tools and Techniques 44
Commenting Out Code: Program Development Tools and Techniques 83
Comments: Program Development Tools and Techniques 23
Completion: Program Development Tools and Techniques 8

Copying Buffers and Files: Program Development Tools and Techniques 63
Creating a File: Program Development Tools and Techniques 9
Definitions: Program Development Tools and Techniques 40
Deriving Methods for Tools and Techniques 3
Documentation: Program Development Tools and Techniques 42
Editing Code: Program Development Tools and Techniques 56
Entering Zmacs: Program Development Tools and Techniques 9
Expanding Macros: Program Development Tools and Techniques 100
Features Described in Tools and Techniques 4
File Attribute Lists: Program Development Tools and Techniques 10
Functions: Program Development Tools and Techniques 40
General Information on Flavors: Program Development Tools and Techniques 141
Getting Started: Program Development Tools and Techniques 9
HELP: Program Development Tools and Techniques 7
Identifying Changed Code: Program Development Tools and Techniques 56
Init Keywords: Program Development Tools and Techniques 144
Introduction to Tools and Techniques 3
Keyboard Macros: Program Development Tools and Techniques 64
Killing and Yanking: Program Development Tools and Techniques 60
Lisp Input Editing: Program Development Tools and Techniques 77
Major and Minor Modes: Program Development Tools and Techniques 12
Methods: Program Development Tools and Techniques 142
Moving Text: Program Development Tools and Techniques 59
Moving Through Text: Program Development Tools and Techniques 59
Multiple Buffers: Program Development Tools and Techniques 65
Names: Program Development Tools and Techniques 41
Objects: Program Development Tools and Techniques 35
Organization of Tools and Techniques 4
Other Displays: Program Development Tools and Techniques 68
Outlining the Figure: Program Development Tools and Techniques 16
Pathnames: Program Development Tools and Techniques 45
Prerequisites to Tools and Techniques 3
Program Development Tools and Techniques 1
Program Strategy: Program Development Tools and Techniques 13
Purpose of Tools and Techniques 3
Scope of Tools and Techniques 3
Searching and Replacing: Program Development Tools and Techniques 57
Signalling Conditions: Program Development Tools and Techniques 133
Simple Screen Output: Program Development Tools and Techniques 14
Stepping: Program Development Tools and Techniques 94
Symbols: Program Development Tools and Techniques 38
The Compiler Warnings Database: Program Development Tools and Techniques 79
The Debugger: Program Development Tools and Techniques 80
The Inspector: Program Development Tools and Techniques 104
Tracing and Stepping: Program Development Tools and Techniques 92
Tracing: Program Development Tools and Techniques 92
Using Multiple Windows: Program Development Tools and Techniques 65
Using Registers: Program Development Tools and Techniques 63
Variables: Program Development Tools and Techniques 39
Zmacs and Other Windows: Program Development Tools and Techniques 66
Compiler Tools and Their Differences 316
Tools for Compiling Code From the Editor Into Your World 316
Tools for Compiling Files 317
Tools for Compiling Single Functions 318
compiler: **top-level-form** property 315
Controlling the Evaluation of Top-level Forms 314
How the Stream Compiler Handles Top-level Forms 310
The Top-level Function 126
si: ***top-level-transformations*** variable 229
Move To Previous Point (c-m-SPACE) Zmacs command 59

Jump To Saved Position (c-X J) Zmacs command 59
:arg option for **trace** 92
:arg option to **trace** 278
:argpdl option for **trace** 92
:argpdl option to **trace** 277
:both option for **trace** 92
:both option to **trace** 278
:break option for **trace** 92, 98
:break option to **trace** 276
:cond option for **trace** 92
:cond option to **trace** 276
:entrycond option for **trace** 92
:entrycond option to **trace** 276
:entry option for **trace** 92
:entry option to **trace** 277
:entryprint option for **trace** 92
:entryprint option to **trace** 277
:error option for **trace** 92, 98
:error option to **trace** 276
:exitbreak option for **trace** 92, 98
:exitbreak option to **trace** 276
:exitcond option for **trace** 92
:exitcond option to **trace** 276
:exit option for **trace** 92
:exit option to **trace** 278
:exitprint option for **trace** 92
:exitprint option to **trace** 277
:nil option for **trace** 92
Options to **trace** 276
:per-process option for **trace** 92
:print option for **trace** 92
:print option to **trace** 277
:step option for **trace** 92, 94
:step option to **trace** 276
:value option for **trace** 92
:value option to **trace** 278
:wherein option for **trace** 92
:wherein option to **trace** 277
si: ***trace-bar-p*** variable 279
si: ***trace-bar-rate*** variable 279
si: ***trace-columns-per-level*** variable 279
trace-compile-flag variable 279
[ARGPDL] **trace** menu item 92
[Break after] **trace** menu item 92
[Break before] **trace** menu item 92
[Cond after] **trace** menu item 92
[Cond before] **trace** menu item 92
[Cond break after] **trace** menu item 92, 98
[Cond break before] **trace** menu item 92, 98
[Conditional] **trace** menu item 92
[Error] **trace** menu item 92, 98
[Per Process] **trace** menu item 92
[Print] **trace** menu item 92
[Print after] **trace** menu item 92
[Print before] **trace** menu item 92
[Step] **trace** menu item 92, 94
[Untrace] **trace** menu item 92
[Wherein] **trace** menu item 92
Trace (m-X) Zmacs command 92, 94
si: ***trace-old-style*** variable 280
:argpdl **trace** Option 277
:break **trace** Option 276
:cond **trace** Option 276

:entry	**trace** Option 277
:entrycond	**trace** Option 276
:entryprint	**trace** Option 277
:error	**trace** Option 276
:exit	**trace** Option 277
:exitbreak	**trace** Option 276
:exitcond	**trace** Option 276
:exitprint	**trace** Option 277
:per-process	**trace** Option 277
:print	**trace** Option 277
:step	**trace** Option 276
:wherein	**trace** Option 277
	trace output 279
Controlling the Format of	**trace** Output 279
	trace special form 92, 94, 98, 275
	[Trace] System menu item 92, 94
	Tracing 92
	Tracing and Stepping: Program Development Tools and Techniques 92
	Tracing Function Execution 275
	Tracing: Program Development Tools and Techniques 92
Keeping	Track of Lisp Syntax 23
:compile-load compound	transformation 201
:fasload simple	transformation 201
:compile	Transformation of **defsystem** 209
:compile-load	Transformation of **defsystem** 210
:compile-load-init	Transformation of **defsystem** 211
:do-components	Transformation of **defsystem** 210
:fasload	Transformation of **defsystem** 209
:load-bfd	Transformation of **defsystem** 210
:readfile	Transformation of **defsystem** 209
Compile-like	transformations 214
Complex	transformations 214
defsystem	Transformations 201, 210, 211
List of **defsystem**	Transformations 209
Load-like	transformations 214
User-defined	transformations 214, 227
Interaction Between **defsystem**	Transformations and **make-system** 202
	Translations file 218
Clearing the	Trap-on-exit Flag for the Current and All Outer Frames 266
Setting the	Trap-on-exit Flag for the Current and All Outer Frames 266
Toggling the	Trap-on-exit Flag for the Current Frame 266
Debugger Commands That	Trap on Frame Exit 266
:any-tyi method of	**tv:any-tyi-mixin** 129
	tv:any-tyi-mixin flavor 129
:function option for	**tv:choose-variable-values** 133
	tv:choose-variable-values function 126, 133
:edges-from init option for	**tv:essential-window** 116
:expose-p init option for	**tv:essential-window** 116
:minimum-height init option for	**tv:essential-window** 116
:minimum-width init option for	**tv:essential-window** 116
:draw-line method of	**tv:graphics-mixin** 14, 116
	tv:graphics-mixin flavor 116
	tv:list-mouse-buttons-mixin flavor 129
	tv:make-blinker function 133
	tv:make-window function 112, 121, 126, 129
:process init option for	**tv:process-mixin** 129
	tv:process-mixin flavor 129
:blinker-p init option for	**tv:sheet** 116
:change-of-size-or-margins method of	**tv:sheet** 116

:init method of	**tv:sheet** 116
:inside-size method of	**tv:sheet** 116
:refresh method of	**tv:sheet** 116, 129
	tv:sheet flavor 116, 133
	tv:sheet-following-blinker function 133
	tv:window flavor 112, 116
	Two Windows (c-X 2) Zmacs command 65
View	Two Windows (c-X 3) Zmacs command 65
Modified	Two Windows (c-X 4) Zmacs command 65
bin file	type 318
	typep function 141
Proceed	types 80, 133
File	Types of Lisp Source and Compiled Code Files 318
	Types of Patch Files 233
File	types of the patch directory file 236
File	types of the system version-directory file 236

U

si:	**unadvise-1** function 283
	unadvise special form 282
	unadvise-within special form 285
Find	Unbalanced Parentheses (m-X) Zmacs command 26
si:	**unbin-file** function 300
	unbreakon function 98
	uncompile function 319
	unspecial special form 314
	untrace special form 92, 280
	[Untrace] **trace** menu item 92
	Untracing Function Execution 280
	unwind-protect special form 133
	Up Comment Line (m-P) Zmacs command 23
	Update Attribute List (m-X) Zmacs command 10
	Update compiler warnings database 321
	:update-directory Option for **make-system** 226
Set sleep time between	updates Peek command 303
	Updating systems 231
Functions	used inside the Debugger 267
	User-defined transformations 214, 227

V

dbg:	**val** function 268
	:value option for **trace** 92
	:value option to **trace** 278
Debugger functions to return	values in current stack frame 267
Examining	values of instance variables 267
	values variable 98, 276
:operation-handled-p method of **si:**	**vanilla-flavor** 142
:which-operations method of **si:**	**vanilla-flavor** 142
si:	**vanilla-flavor** flavor 142
applyhook	variable 291
arglist	variable 98, 276
compiler:functions-defined	variable 324
compiler:functions-referenced	variable 325
dbg:*debug-io-override*	variable 268
dbg:*defer-package-dwim*	variable 268
dbg:*frame*	variable 268
dbg:*show-backtrace*	variable 268
evalhook	variable 289
ignore	variable 324
query-io	variable 133

si:advised-functions variable 283
si:*batch-mode-p* variable 229
si:*file-transformation-function* variable 229
si:*make-system-forms-to-be-evaled-after* variable 229
si:*make-system-forms-to-be-evaled-before* variable 228
si:*make-system-forms-to-be-evaled-finally* variable 229
si:*query-type* variable 229
si:*redo-all* variable 229
si:*silent-p* variable 229
si:*system-being-defined* variable 214
si:*system-being-made* variable 228
si:*top-level-transformations* variable 229
si:*trace-bar-p* variable 279
si:*trace-bar-rate* variable 279
si:*trace-columns-per-level* variable 279
si:*trace-old-style* variable 280
standard-output variable 75
sys:file-local-declarations variable 324
terminal-io variable 14, 112
trace-compile-flag variable 279
values variable 98, 276
Describe Variable At Point (c-sh-V) Zmacs command 39
Rebound Variable Bindings During Evaluation 260
Overriding Variable-defined-but-never-referenced Warnings 326
Variables 39
Compiler variables 324
Debugger Variables 268
Examining values of instance variables 267
Instance variables 113, 124, 126
Variables: Program Development Tools and Techniques 39
:verbose Option for **load-patches** 245, 246
Major version 236
Minor version 236
File types of the system version-directory file 236
System Version-directory File 233
Major version number 231
Minor version number 231, 238
:latest symbol in **make-system** **:version** option 222
:newest symbol in **make-system** **:version** option 222
:released symbol in **make-system** **:version** option 222
:version Option for **make-system** 223
System versions 231
View Directory (m-X) Zmacs command 45
View Patches (m-X) 242
View Two Windows (c-X 3) Zmacs command 65
Set Vsp (m-X) Zmacs command 10

W W W

Compiler warnings 65, 70, 79
Compiler Style Warnings 323
Controlling Compiler Warnings 323
Function-referenced-but-never-defined Warnings 324
Overriding Variable-defined-but-never-referenced Warnings 326
Compiler Warnings Database 321
Print compiler warnings database 321
Update compiler warnings database 321
The Compiler Warnings Database: Program Development Tools and Techniques 79
Compiler Warnings (m-X) Zmacs command 79, 321
Edit Compiler Warnings (m-X) Zmacs command 79, 321
Edit File Warnings (m-X) Zmacs command 321

Load Compiler	Warnings (m-X) Zmacs command 79, 321
	what-files-call function 38
	:wherein option for **trace** 92
	:wherein option to **trace** 277
	[Wherein] **trace** menu item 92
	:wherein trace Option 277
	where-is function 38
	Where Is Symbol (m-X) Zmacs command 38
	:which-operations method of **si:vanilla-flavor** 142
	who-calls function 38
Mark	Whole (c-X H) Zmacs command 63
	:who-line-documentation-string method 129
	Whoppers 116
Set Tab	Width (m-X) Zmacs command 10
Choose-variable-values	window 126, 129
The Basic Arrow	Window 116
Display status of	window area 303
Scroll Other	Window (c-m-V) Zmacs command 65
One	Window (c-X 1) Zmacs command 65
Other	Window (c-X O) Zmacs command 65
tv:	**window** flavor 112, 116
The Arrow	Window: Interaction, Processes, and the Mouse 129
Multiple	windows 65
Programming Aids for Flavors and	Windows 141
Using Flavors and	Windows 111
Two	Windows (c-X 2) Zmacs command 65
View Two	Windows (c-X 3) Zmacs command 65
Modified Two	Windows (c-X 4) Zmacs command 65
Using Multiple	Windows: Program Development Tools and Techniques 65
Zmacs and Other	Windows: Program Development Tools and Techniques 66
Advising One Function	Within Another 285
	with-open-stream special form 126
Atom	Word Mode (m-X) Zmacs command 12
How the Inspector	Works 295
Tools for Compiling Code From the Editor Into Your	World 316
	Writing and Editing Code 7

X

Put Register (c-X	X) Zmacs command 63

Y

	Yank (c-Y) Zmacs command 60
Killing and	Yanking: Program Development Tools and Techniques 60
	Yanking text 60
	Yank Pop (m-Y) Zmacs command 60
Before	You Begin: Program Development Tools and Techniques 7
Tools for Compiling Code From the Editor Into	Your World 316

Z

	Zmacs and Other Windows: Program Development Tools and Techniques 66
Compiling Code in a	Zmacs Buffer 70
Summary of Compiler Actions on Code in a	Zmacs Buffer 71
Atom Word Mode (m-X)	Zmacs command 12

Auto Fill Mode (m-X) Zmacs command 12
Backward Kill Sexp (c-m-RUBOUT) Zmacs command 60
Beep (c-G) Zmacs command 59
Brief Documentation (c-sh-D) Zmacs command 39, 42
Call Last Kbd Macro (c-X E) Zmacs command 64
Compile Buffer (m-X) Zmacs command 70
Compile Changed Definitions (m-X) Zmacs command 70
Compile Changed Definitions Of Buffer (m-sh-C) Zmacs command 70
Compile File (m-X) Zmacs command 73
Compile Region (c-sh-C) Zmacs command 70
Compile Region (m-X) Zmacs command 316
Compiler Warnings (m-X) Zmacs command 79, 321
c-sh-C Zmacs command 316
Deinstall Macro (m-X) Zmacs command 64
Describe Flavor (m-X) Zmacs command 141
Describe Variable At Point (c-sh-V) Zmacs command 39
Dired (m-X) Zmacs command 45
Disassemble (m-X) Zmacs command 104
Display Directory (c-X c-D) Zmacs command 45
Down Comment Line (m-N) Zmacs command 23
Edit Callers (m-X) Zmacs command 44
Edit Changed Definitions (m-X) Zmacs command 56
Edit Changed Definitions Of Buffer (m-X) Zmacs command 56
Edit Combined Methods (m-X) Zmacs command 142
Edit Compiler Warnings (m-X) Zmacs command 79, 321
Edit Definition (m-.) Zmacs command 40, 141, 142
Edit File Warnings (m-X) Zmacs command 321
Edit Methods (m-X) Zmacs command 65, 142
Electric Shift Lock Mode (m-X) Zmacs command 12
End Kbd Macro (c-X)) Zmacs command 64
Evaluate And Replace Into Buffer (m-X) Zmacs command 75
Evaluate Buffer (m-X) Zmacs command 75
Evaluate Changed Definitions (m-X) Zmacs command 75
Evaluate Changed Definitions Of Buffer (m-sh-E) Zmacs command 75
Evaluate Into Buffer (m-X) Zmacs command 75
Evaluate Minibuffer (m-ESCAPE) Zmacs command 75
Evaluate Region (c-sh-E) Zmacs command 75
Fill Long Comment (m-X) Zmacs command 23
Find File (c-X c-F) Zmacs command 9
Find Unbalanced Parentheses (m-X) Zmacs command 26
Function Apropos (m-X) Zmacs command 41
HELP Zmacs command 7, 64
Incremental Search (c-S) Zmacs command 57
Indent For Comment (c-; or m-;) Zmacs command 23
Indent For Lisp (TAB or c-m-TAB) Zmacs command 26
Indent New Comment Line (m-LINE) Zmacs command 23
Indent New Line (LINE) Zmacs command 26
Indent Region (c-m-\) Zmacs command 26
Indent Sexp (c-m-Q) Zmacs command 26
Insert Buffer (m-X) Zmacs command 63
Insert File (m-X) Zmacs command 63
Install Macro (m-X) Zmacs command 64
Install Mouse Macro (m-X) Zmacs command 64
Jump To Saved Position (c-X J) Zmacs command 59
Kill Comment (c-m-;) Zmacs command 23
Kill Sexp (c-m-K) Zmacs command 60
Lisp Mode (m-X) Zmacs command 12
List Callers (m-X) Zmacs command 38, 44
List Changed Definitions (m-X) Zmacs command 56
List Changed Definitions Of Buffer (m-X) Zmacs command 56
List Combined Methods (m-X) Zmacs command 142
List Matching Lines (m-X) Zmacs command 57
List Matching Symbols (m-X) Zmacs command 38

List Methods (m-X) Zmacs command 142
Load Compiler Warnings (m-X) Zmacs command 79, 321
Load File (m-X) Zmacs command 73
Long Documentation (m-sh-D) Zmacs command 39, 42
Macro Expand Expression All (m-X) Zmacs command 100
Macro Expand Expression (c-sh-M) Zmacs command 100
Mark Definition (c-m-H) Zmacs command 60
Mark Whole (c-X H) Zmacs command 63
Modified Two Windows (c-X 4) Zmacs command 65
Move To Previous Point (c-m-SPACE) Zmacs command 59
Multiple Edit Callers (m-X) Zmacs command 44
Multiple List Callers (m-X) Zmacs command 44
Name Last Kbd Macro (m-X) Zmacs command 64
One Window (c-X 1) Zmacs command 65
Open Get Register (c-X G) Zmacs command 63
Other Window (c-X O) Zmacs command 65
Print Modifications (m-X) Zmacs command 56
Push Pop Point Explicit (m-SPACE) Zmacs command 59
Put Register (c-X X) Zmacs command 63
Query Replace (m-%) Zmacs command 57
Quick Arglist (c-sh-A) Zmacs command 43
Quit (c-Z) Zmacs command 80
Reparse Attribute List (m-X) Zmacs command 10
Replace (c-%) Zmacs command 57
Reverse Search (c-R) Zmacs command 57
Save Position (c-X S) Zmacs command 59
Save Region (m-W) Zmacs command 60
Scroll Other Window (c-m-V) Zmacs command 65
Select All Buffers As Tag Table (m-X) Zmacs command 57
Select Buffer (c-X B) Zmacs command 59
Select Previous Buffer (c-m-L) Zmacs command 59
Select System As Tag Table (m-X) Zmacs command 57
Set Backspace (m-X) Zmacs command 10
Set Base (m-X) Zmacs command 10
Set Comment Column (c-X ;) Zmacs command 23
Set Fill Column (c-X F) Zmacs command 12
Set Fonts (m-X) Zmacs command 10
Set Key (m-X) Zmacs command 64
Set Lowercase (m-X) Zmacs command 10
Set Nofill (m-X) Zmacs command 10
Set Package (m-X) Zmacs command 10
Set Patch File (m-X) Zmacs command 10
Set Pop Mark (c-SPACE) Zmacs command 59
Set Tab Width (m-X) Zmacs command 10
Set Vsp (m-X) Zmacs command 10
Source Compare Merge (m-X) Zmacs command 56
Source Compare (m-X) Zmacs command 56
Split Screen (m-X) Zmacs command 65
Start Kbd Macro (c-X (Zmacs command 64
Swap Point And Mark (c-X c-X) Zmacs command 59
Tags Query Replace (m-X) Zmacs command 57
Tags Search (m-X) Zmacs command 57
Trace (m-X) Zmacs command 92, 94
Two Windows (c-X 2) Zmacs command 65
Up Comment Line (m-P) Zmacs command 23
Update Attribute List (m-X) Zmacs command 10
View Directory (m-X) Zmacs command 45
View Two Windows (c-X 3) Zmacs command 65
Where Is Symbol (m-X) Zmacs command 38
Yank (c-Y) Zmacs command 60
Yank Pop (m-Y) Zmacs command 60
↑r Dired (c-X D) Zmacs command 45
c-? Zmacs minibuffer command 8

COMPLETE Zmacs minibuffer command 8
END Zmacs minibuffer command 8
HELP Zmacs minibuffer command 8
RETURN Zmacs minibuffer command 8
SPACE Zmacs minibuffer command 8
Entering Zmacs: Program Development Tools and Techniques 9
zwei:*send-mail-about-patch* 242

\ \ \

[Set \] Inspector menu item 298

↑ ↑ ↑

↑r Dired (c-X D) Zmacs command 45