symbolics™

# Program Development Tools and Techniques

August 1983

Cambridge, Massachusetts

# Program Development Tools and Techniques

**# 990001**

**August 1983**

**This document corresponds to Release 4.5.**

This document was prepared by the Documentation and Education Services Department of Symbolics, Inc. Principal writer(s): Robert Mathews and Sherry Finkel.

# Table of Contents

# List of Figures



# List of Tables

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

- *Lisp Machine Summary*
- *Program Development Help Facilities*
- *Lisp Machine Manual*
- An up-to-date set of release notes

## 1.3 Scope

We focus in this document on interaction between programmers and the Lisp Machine. We present some ways of using 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. Some key bindings and symbol names are different on the Symbolics LM-2.

## 1.4 Method

We derived the methods we describe here by working with Lisp Machine programmers at Symbolics. Some of these programmers were early developers of the 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 Lisp Machine to accommodate a relatively spontaneous and incremental programming style. Five Lisp Machine features make up an integrated programming environment.

- *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 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 he 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. Chapter 5 presents some simple uses of flavors and windows and some programming aids for working with them.

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. The complete code appears in appendixes A (page 133) and B (page 147) and in the files SYS: EXAMPLES; ARROW-CALC LISP and SYS: EXAMPLES; ARROW-OUT LISP. (To run the program, load SYS: EXAMPLES; ARROW.) A reproduction of the design produced on a Symbolics LGP-1 Laser Graphics Printer appears in appendix C (page 165).

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

## 1.7 Notation Conventions

Modifier keys are designed to be held down while pressing other keys. They do not themselves transmit characters. A combined keystroke like META-X is pronounced "meta x" and written as m-X. This notation means press the META key and, while holding it down, press the X key.

Modifier keys are abbreviated as follows:

| *Key* | *Abbreviation* |
|---|---|
| CTRL | c- |
| META | m- |
| SUPER | s- |
| HYPER | h- |
| SHIFT | sh- |

The keys with white lettering (like X or SELECT) all transmit characters. Combinations of these keys are meant to be pressed in sequence. This sequence is written as, for example, SELECT L. This notation means press the SELECT key, release it, and then press the L key.

This document uses the following notation conventions:

| *Appearance in document* | *Representing* |
|---|---|
| **send, chaos:host-up** | Printed representation of Lisp objects in running text. |
| RETURN, ABORT, c-F | Keyboard keys. |
| SPACE | Space bar. |
| `login` | Literal type-in. |
| `(make-symbol "foo")` | Lisp code examples. |
| (**function-name** *arg1* *arg2*) | Syntax description of the invocation of **function-name**. |
| *arg1* | Argument to the function **function-name**, usually expressed as a word that reflects the type of argument (e.g., *string*). |
| *arg2* | Optional argument; you can leave it out. |
| Undo, Tree Edit Any | Command names in Zmacs and Zmail appear with initial letter of each word capitalized. |
| Insert File (m-X) | Extended command names in Zmacs and Zmail. Use m-X to invoke one. |
| [Map Over] | Menu items. |
| (L), (R2) | Mouse clicks: L=left, L2=double click left, M=middle, M2=double click middle, R=right, R2=double click right. |

Mouse commands use notations for menu items and mouse clicks in the following ways:

1. Square brackets delimit a mouse command.
2. Slashes (/) separate the members of a compound mouse command.
3. The standard clicking pattern is as follows:

- For a single menu item, always click left. For example, the following two commands are exactly the same:

  [Previous]
  [Previous (L)]

- For a compound command, always click right on each menu item except the last, where you click left. For example, the following two compound commands are exactly the same:

  [Map Over / Move / Hardcopy]
  [Map Over (R) / Move (R) / Hardcopy (L)]

4. When the notation does not follow the standard, it shows explicitly which button to click. For example:

   [Map Over / Move (M)]
   [Previous (R)]

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 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 document. (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 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

Appendixes A (page 133), B (page 147), and C (page 165) contain the code for the sample program and a reproduction of the LGP image the program produces.

## 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. (See *Program Development Help Facilities* for details.)

### 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 ? | 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 c 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 or [Edit] from a system menu to enter Zmacs.

**Reference**

| | |
|---|---|
| SELECT E | Selects a Zmacs frame. |
| [Edit] (from a system menu) | 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 section 2.2.3, page 7). 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**.

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 1,000 symbols, type into the attribute list:

```
Package: (GRAPHICS GLOBAL 1000)
```

See the *Lisp Machine Manual*, section 21.9.2, page 369, and *Release 4.0 Release Notes*, sections 5.3.5 and 5.3.6, page 90, for more on file and buffer attributes.

---

**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, 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 section 2.4, page 19).

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
  (setq 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 upper case. |
| 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: The first appears in appendix A (page 133), the second in appendix B (page 147).

How do we send line positions from the module that calculates them to the module that transmits them to output? The output module, which we discuss in detail in chapter 5 (page 101), consists of definitions of flavors and methods to transfer information to the appropriate output stream. 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

We discuss the output routines in chapter 5 (page 101). 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. (See *Introduction to Using the Window System*, section 2.4, page 30, for more on **:draw-line**.)

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. As the diagram in appendix A (page 133) shows, seven points define each arrow. 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 *Lisp Machine Manual*, chapter 19, page 257).
- 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 *Lisp Machine Manual*, chapter 20, page 279).
- 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 section 5.1.6, page 117).

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 (see the *Lisp Machine Manual*, section 3.1, page 14). This macro 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 section 3.1, page 62). 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 (page 18) shows the output of the program at this stage.

These six steps illustrate a pattern of incremental program development:

- We make each function initially simple. We add new functions and edit old ones as tasks become more complex or refined. Facilities for keeping track of Lisp syntax (section 2.4, page 19) and for editing code (section 2.8, page 49) encourage this incremental style.
- 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.

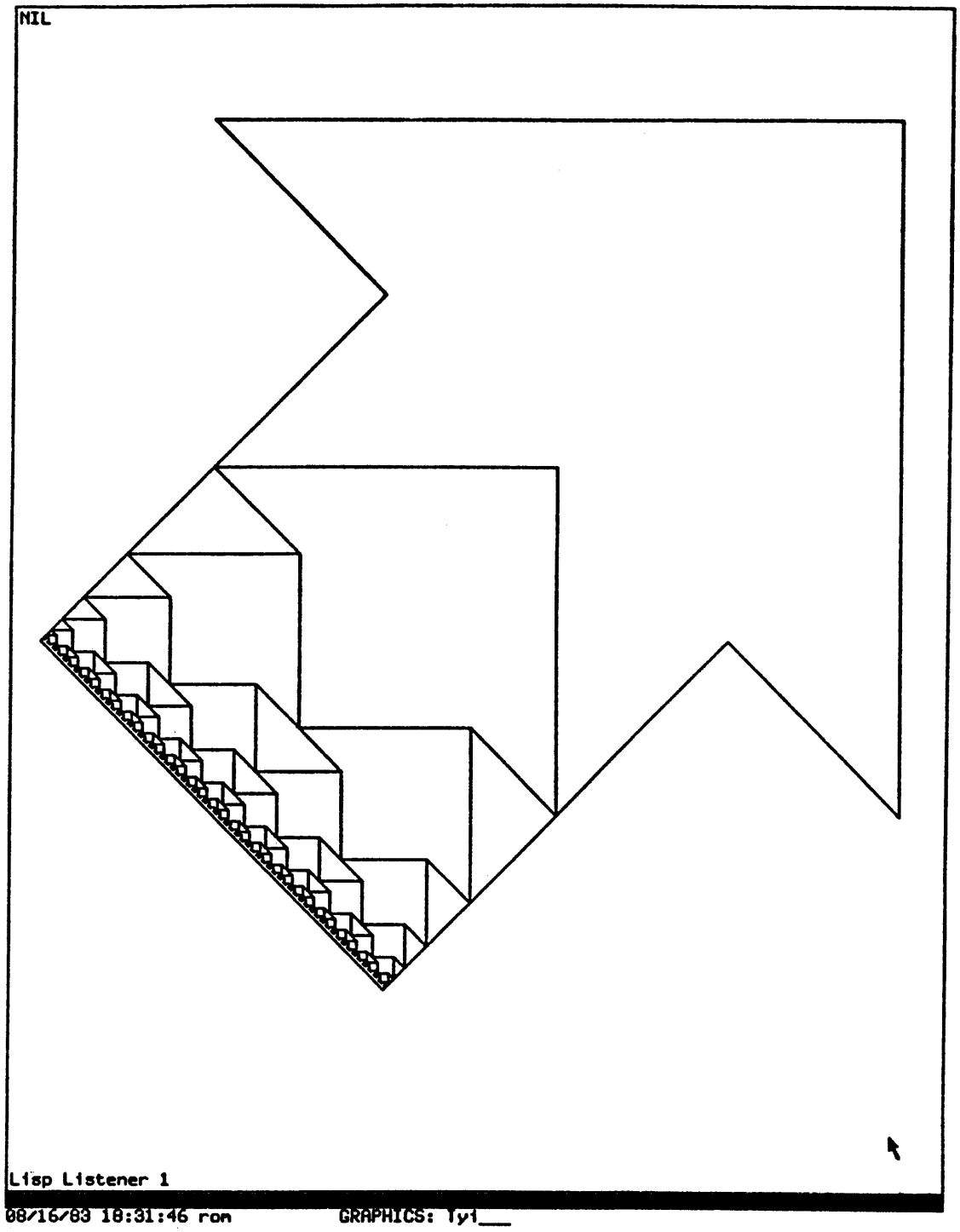



Figure 1. Program output with only the outlines of the arrows in the figure.

To support this incremental style, we must be able to check the syntax of our code, edit it, and compile it in sections. We discuss facilities for keeping track of Lisp syntax in the next section; techniques for editing code in section 2.8 (page 49); and methods of compiling and evaluating in chapter 3 (page 61).

## 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 section 2.2.4, page 9).

See the *Lisp Machine Summary* for a list of common Zmacs commands for operating on Lisp expressions.

### 2.4.1 Comments

You can document code in two ways: You can supply documentation strings for functions, variables, and constants (see section 2.6, page 28); and you can insert comments in the source code. You can retrieve documentation strings with Zmacs commands and Lisp functions (section 2.6, page 28). 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 chapter 3, page 61, and chapter 4, page 71).

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.

You can also use #| to begin a comment and |# to end one. The comment can extend for more than one line. You can nest #| and |# within longer comments.

## 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
  (setq 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 (page 29) 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 (section 2.6, page 28).
- We must start to change existing code, adding function calls and new arguments. These changes require increasing use of facilities for editing code (section 2.8, page 49).

## 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 section 4.7, page 94).

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

#### Example

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

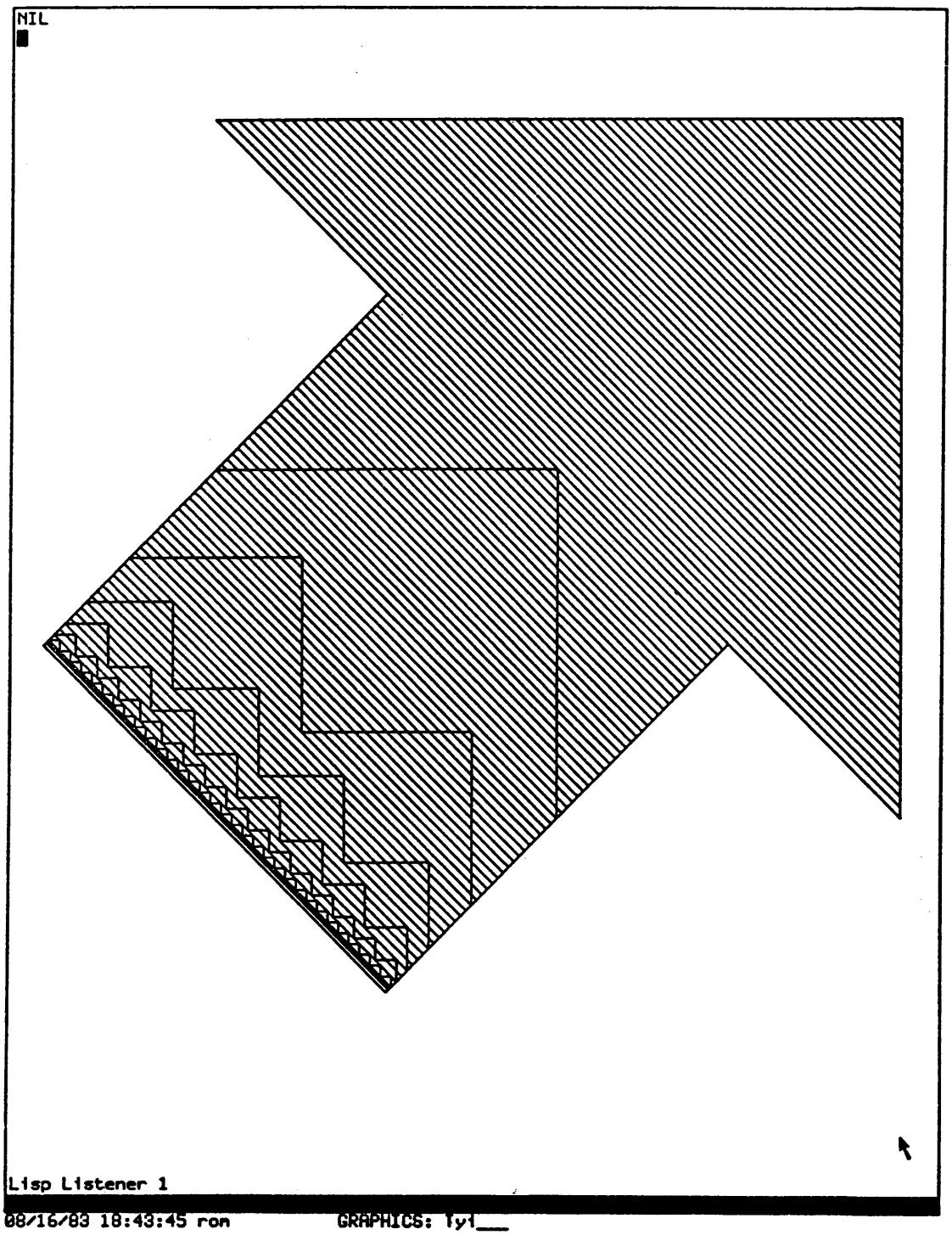



Figure 2. Program output with stripes of even spacing and density.

```
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. (See *Program Development Help Facilities* for more on the meanings and default values of arguments to these functions.)

---

**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 section 2.6.4, page 33).

---

**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 section 2.6.4, page 33).

**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.

Section 5.2.2 (page 129) describes how to use Edit Definition (m-.) to edit definitions of flavor methods.

**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 string 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 first line of the documentation string:

```
DRAW-ARROW-GRAPHIC: Function controlling the calculation module.
```

To ensure that c-sh-D displays meaningful information, make the first line of each documentation string a complete sentence that summarizes the function.

---

**Reference**

| | |
|---|---|
| Brief Documentation (c-sh-D) | Displays the first line of the function's documentation string. |

| | |
|---|---|
| Long Documentation (m-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; see the *Lisp Machine Manual*, section 10.9, page 150, and *Program Development Help Facilities* for details.

---

**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. |
| tr 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 (page 48) shows the output of the program with stripes of varying spacing and thickness.

At this stage in developing the program we define new functions, constants, and variables. But most of the work consists of changing existing code. Often you need to make similar changes

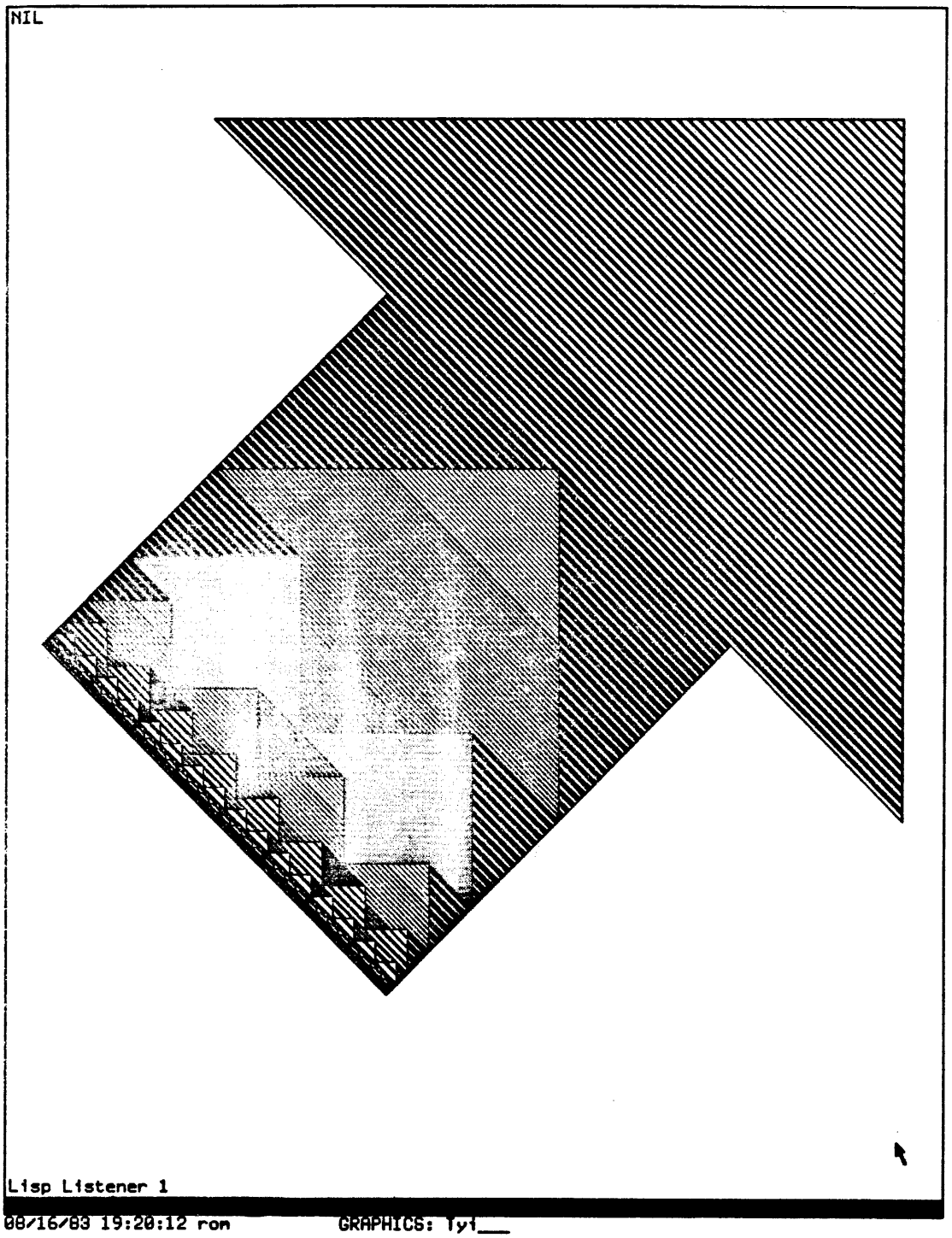



Figure 3. Program output with stripes of varying spacing and density.

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 Lisp Machine facilities are those for finding out about existing code (section 2.6, page 28) and for editing code (section 2.8, page 49).

## 2.8 Editing Code

The features we discussed in sections 2.2 (page 7) and 2.4 (page 19) are useful mainly in composing new code. The features we described in section 2.6 (page 28) are helpful in both writing and editing code. 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

Many of the facilities discussed in section 2.6 (page 28), particularly the series of List and Edit commands, are useful for displaying and moving to code you wish to edit. The commands we discuss here find and replace strings. *Tags 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 tags table for all buffers read in. Use Select System As Tag Table (m-X) to create a tags table for all files in a system. (For information on systems, see the *Lisp Machine Manual*, chapter 24, page 406.)

**Example**

We have defined ***stripe-d***, and we want to replace some occurrences of the constant ***stripe-distance*** by the variable ***stripe-d***. We use Query Replace (m-Z) to find each occurrence of ***stripe-distance***. By pressing SPACE, we replace ***stripe-distance*** by ***stripe-d*** in functions like **stripe-arrowhead**. By pressing RUBOUT, we leave ***stripe-distance*** in place in functions like **draw-big-arrow-shaft-stripes-left**.

**Reference**

| | |
|---|---|
| List Matching Lines (m-X) | Displays the lines (following point) in the current buffer that contain a string. |
| Incremental Search (c-S) | Prompts for a string and moves forward to its first occurrence in the buffer. Press c-S to repeat the search with the same string. Press c-R to search backward with the same string. After you invoke the command, if c-S is the first character you type (instead of a string), uses the string specified in the previous search. |
| Reverse Search (c-R) | Prompts for a string and moves backward to its last occurrence in the buffer. Press c-R to repeat the |

| | |
|---|---|
| | search with the same string. Press c-S to search forward with the same string. After you invoke the command, if c-R is the first character you type (instead of a string), uses the string specified in the previous search. |
| Tags Search (m-X) | Searches for a string in all files listed in a tags table. |
| Replace (c-Z) | In the buffer, replaces all occurrences (following point) of one string by another. |
| Query Replace (m-Z) | 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 tags table, replaces occurrences of one string by another, querying before each replacement. |
| Select All Buffers As Tag Table (m-X) | Creates a tags table for all buffers in Zmacs. |
| Select System As Tag Table (m-X) | Creates a tags 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 anew. 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 1)
                 (- end-x 12) 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 1)
                 (- end-x 12) 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 1)
                 (- end-x 12) end-y)
        (send *dest* ':show-lines first-x (- start-y 1)
                 (- end-x 12) 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 1)
                 (- end-x 12) end-y)
        (send *dest* ':show-lines first-x (- start-y 1 1)
                 (- end-x 12 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 double-middle to push it onto the kill ring. Move to the new buffer. Click double-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:<br>• First repetition: kills the text of *region*, pushing the text onto the kill ring (like c-W)<br>• Second repetition: pops the text of *region* from the kill ring, inserting the text into the buffer at point (like c-Y)<br>• 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)<br>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

### 2.8.5.5 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 section 4.1, page 71).

**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 section 5.2.2, page 129).

**Example**

After finishing the program, we collect a file of bug reports from users. We want to use these reports in correcting our program code. We create two windows, one displaying the program code and the other the bug-report file. We edit the program code, using c-m-V to scroll the bug-report window as we correct each bug.

**Reference**

| | |
|---|---|
| Split Screen (m-X) | Pops up a menu of buffers and splits the screen to display the buffers you select. |
| Two Windows (c-X 2) | Creates a second window, with the current buffer on top and the previous buffer on the bottom. Puts the cursor in the bottom window. |
| View Two Windows (c-X 3) | Creates a second window, with the current buffer on top and the previous buffer on the bottom. Puts the cursor in the top window. |
| Modified Two Windows (c-X 4) | Creates a second window and visits a buffer, file, or tag there. Displays the current buffer in the top window. |
| Other Window (c-X 0) | Moves to the other of two windows. |
| Scroll Other Window (c-m-V) | Scrolls the other of two windows. |
| One Window (c-X 1) | Returns to one-window display, selecting the buffer the cursor is in. |

#### 2.8.5.6 Zmacs and Other Windows

Use [Split Screen] or [Edit Screen] from a 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 a 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 (page 59) 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 a system menu)
Adds a Lisp Listener to a screen displaying an existing Zmacs buffer.

[Split Screen / Existing Window / *Existing Zmacs Buffer* / Do It] (from a system menu)
Resumes one-window display of an existing Zmacs buffer.

#### 2.8.5.7 Other Displays

The window system allows you to use menus, choose-variable-values windows, and other multiple-window displays in executing programs. See *Introduction to Using the Window System* and *Lisp Machine Choice Facilities* for details. See chapter 5 (page 101) for examples of simple uses of windows, including choose-variable-values windows.

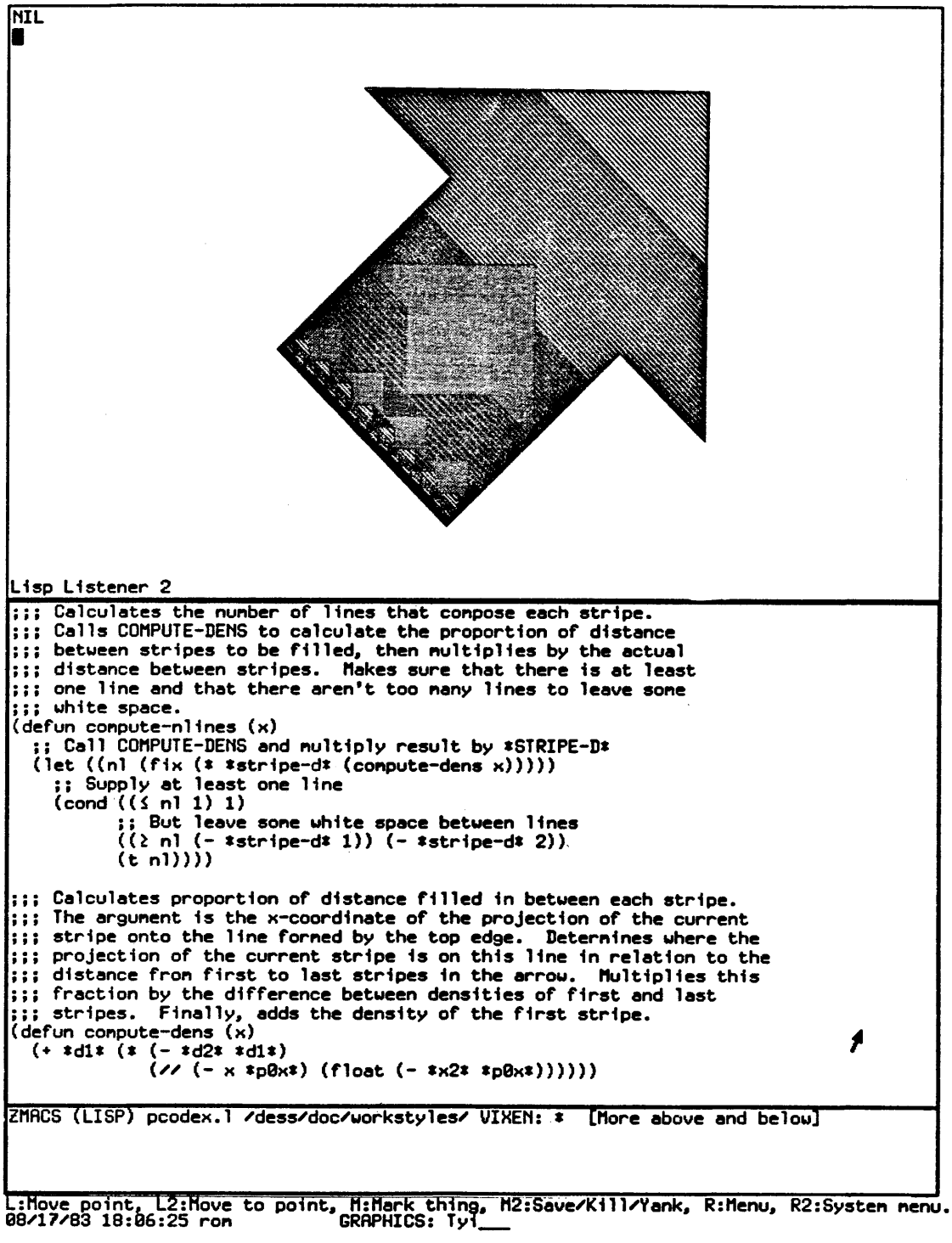



Figure 4. Using multiple windows to test the program while viewing the source code.

# 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. We summarize the compiler's actions in section 3.1.1 (page 62).
- 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). We discuss compiling files in section 3.1.2 (page 65).

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.

See the *Lisp Machine Manual*, chapter 10, page 136, for more information on functions.

## 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 section 2.6, page 28).

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. Chapter 4 (page 71) describes how Edit Compiler Warnings (m-X) lets you use the compiler warnings as a reference source for debugging.

See the *Lisp Machine Manual*, chapter 16, page 197, for more information on the compiler.

### 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. (See the *Lisp Machine Manual*, chapter 16, page 197, for details.) |
| 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 to **trace** (see section 4.4.2, page 86). 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 a system menu to select a Lisp Listener and evaluate forms there. Use SELECT E or [Edit] from a 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 section 4.4.2, page 86).

---

**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-C` to yank the last form we typed. If that is not `(do-arrow)`, we press `m-C` until `(do-arrow` appears, without the close parenthesis. We type a close parenthesis to begin the evaluation.

**Reference**

| | |
|---|---|
| `c-C` | Yanks the last form typed to the Lisp Listener. Excludes the final delimiter, allowing you to edit the form before evaluating it. With an argument *n*, yanks the *n*th form in the input ring. (This command is not available in Zmacs.) |
| `m-C` | After a `c-C` 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. (This command is not available in Zmacs.) |

# 4. Debugging Lisp Programs

The Lisp Machine 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 *Lisp Machine Manual*, section 24.3, page 411). 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**

In section 3.1.1 (page 62), we discussed compiling the initial code for the calculation module of the sample program. Figure 5 (page 72) 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.

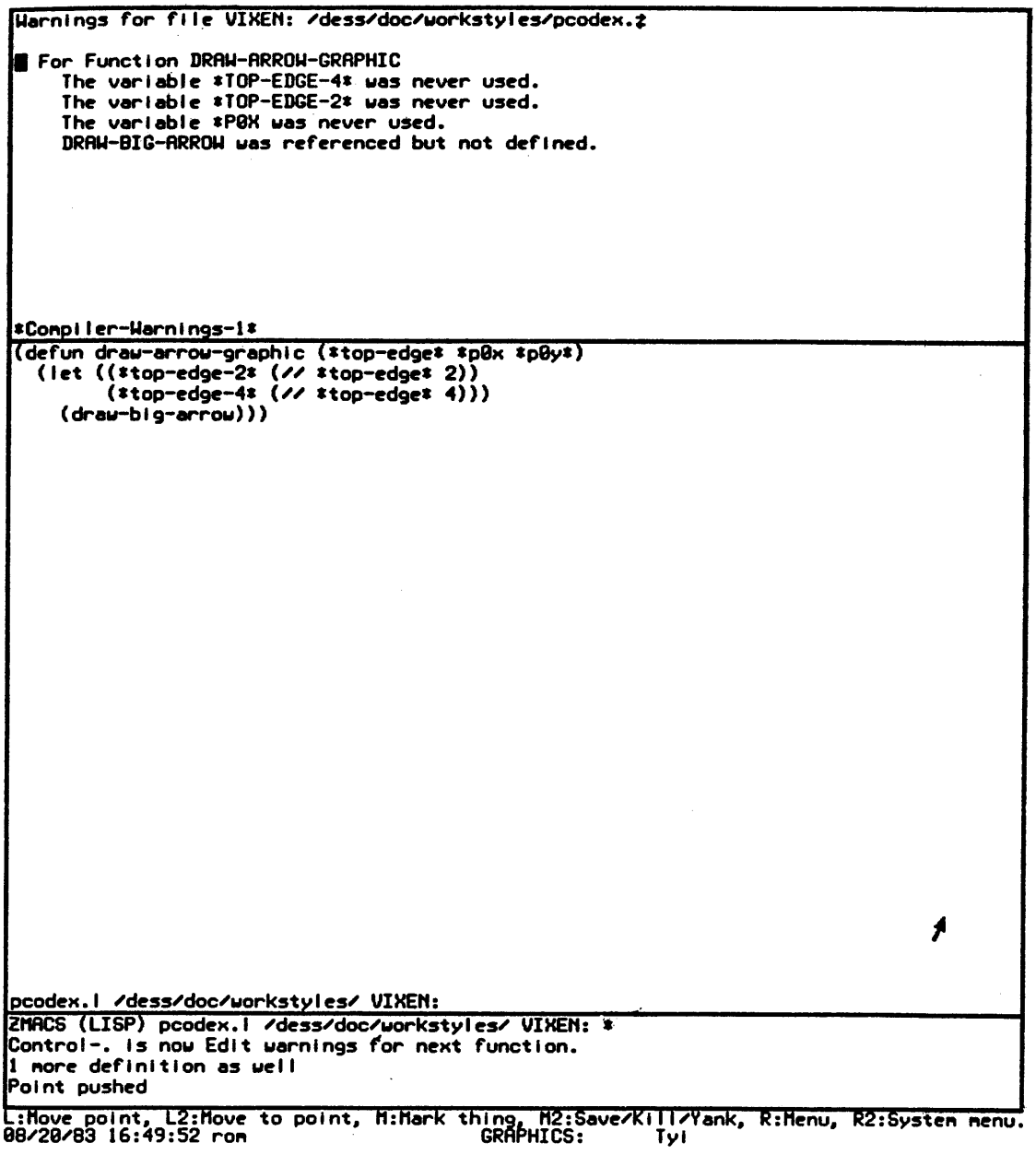



Figure 5. Edit Compiler Warnings (m-X) splits the screen. The upper window contains compiler warnings. The lower window contains the source code.

| 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 section 4.5, page 89.)

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 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 *Signalling and Handling Conditions*, section 10.5, page 41). 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 (page 76) 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 (page 77) 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 **#|** and **|#**.

**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**:

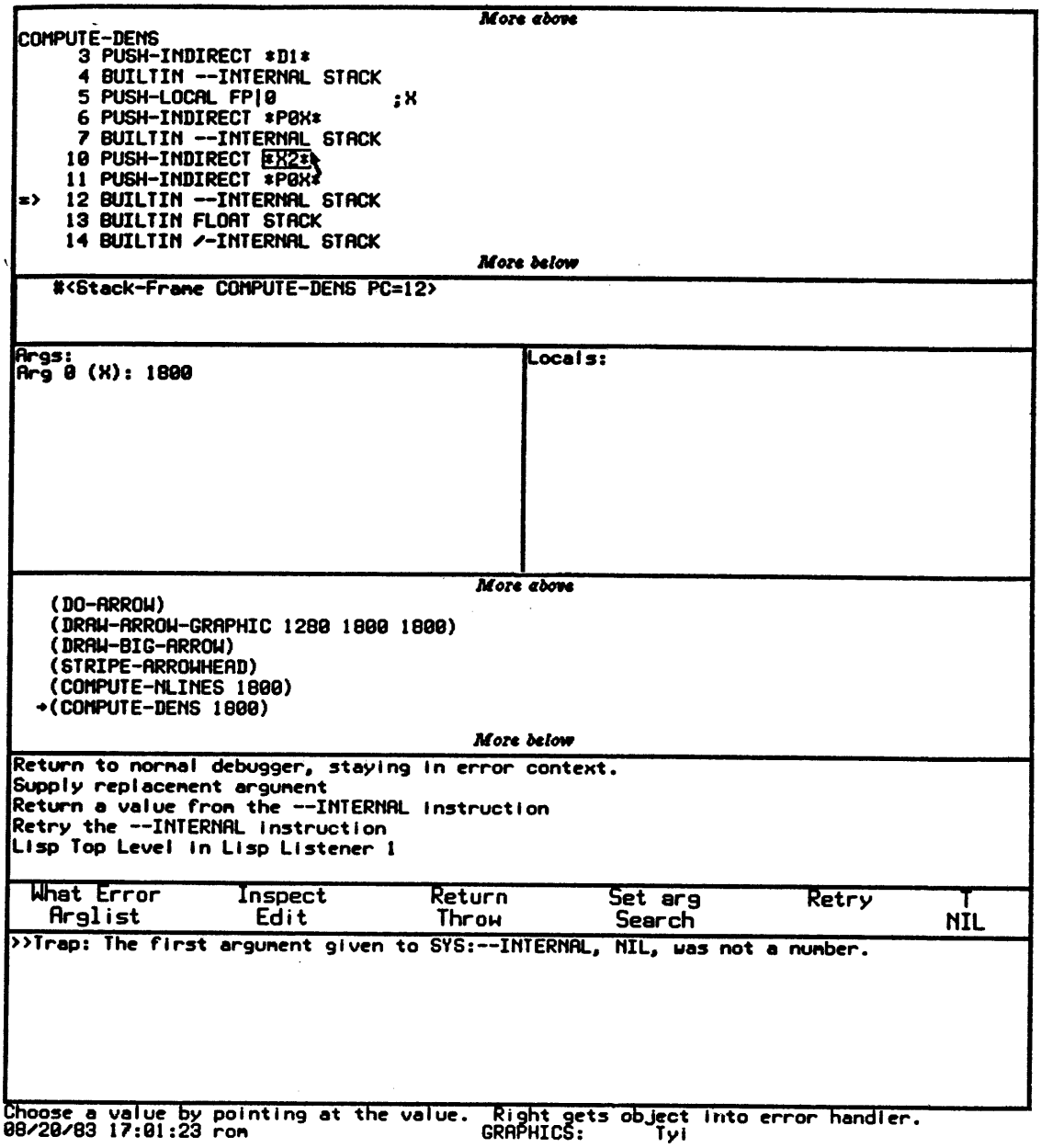



Figure 6. The Display Debugger: inspecting the stack frame containing a call to **compute-dens**.

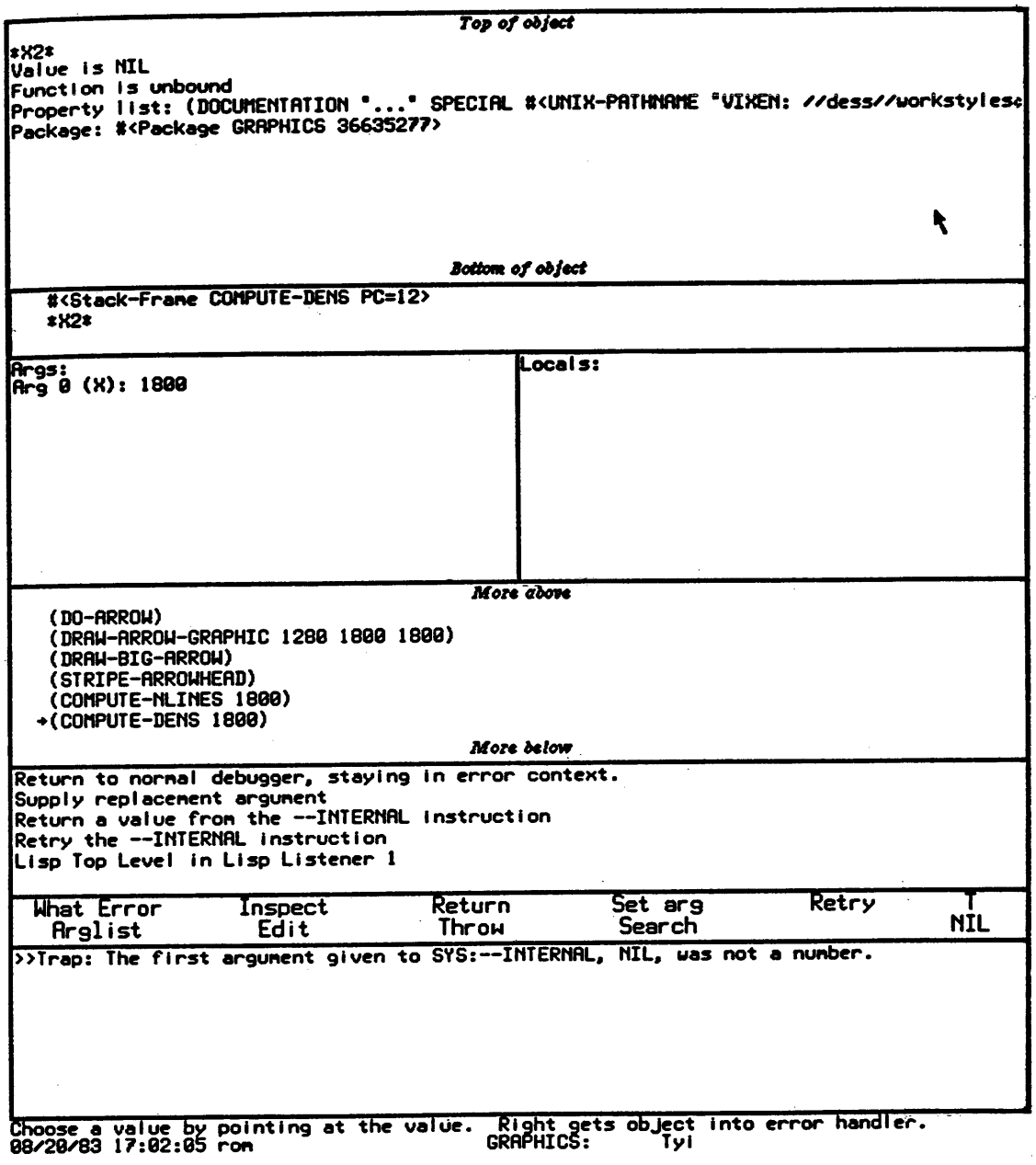



Figure 7. The Display Debugger: inspecting the variable ***x2***.

```
(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 (page 81). 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 (page 82). 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 (page 83). 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 (page 18).

---

### Example

Figure 4 (page 59) 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))
```

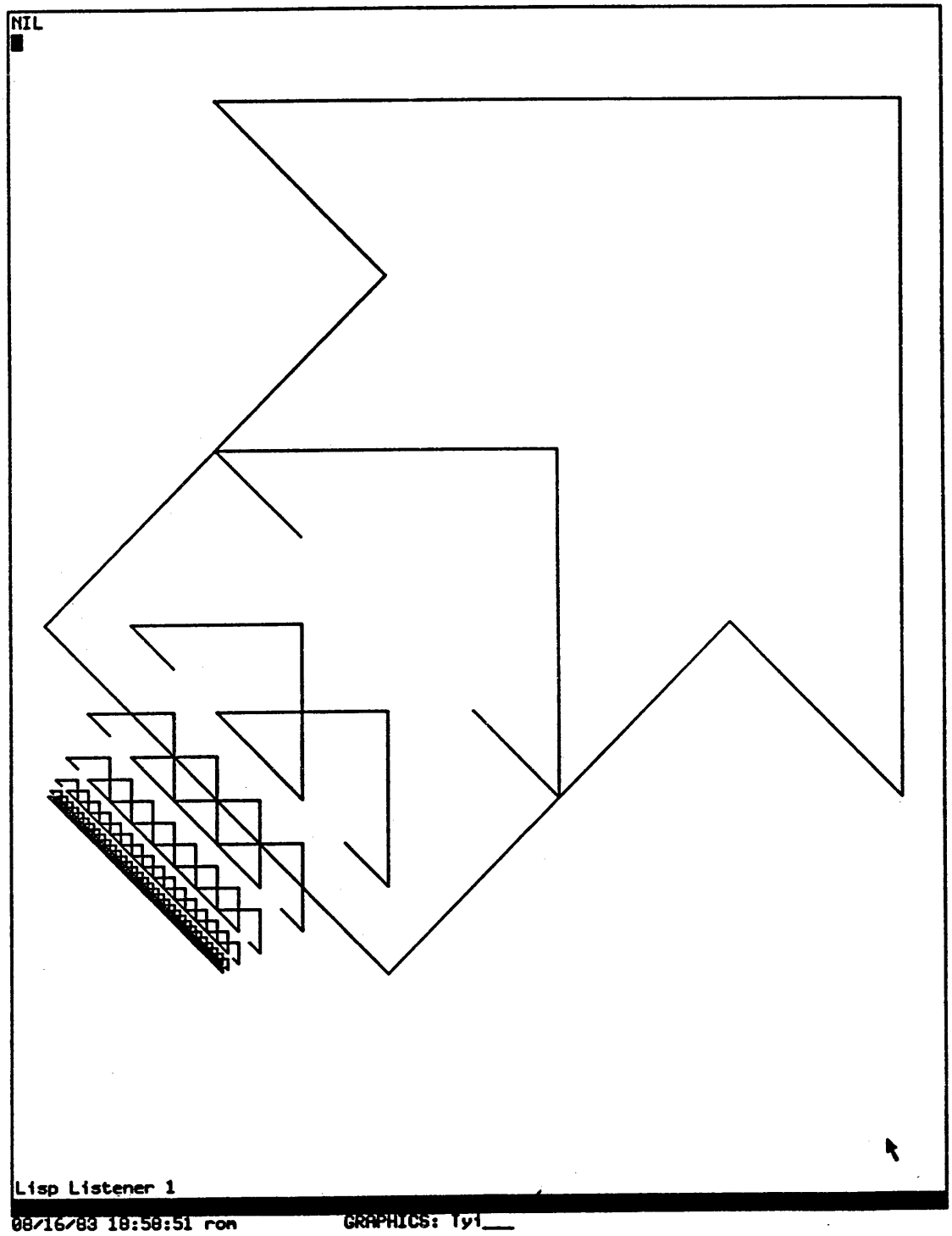



Figure 8. Output resulting from a faulty attempt to outline the small arrows recursively.

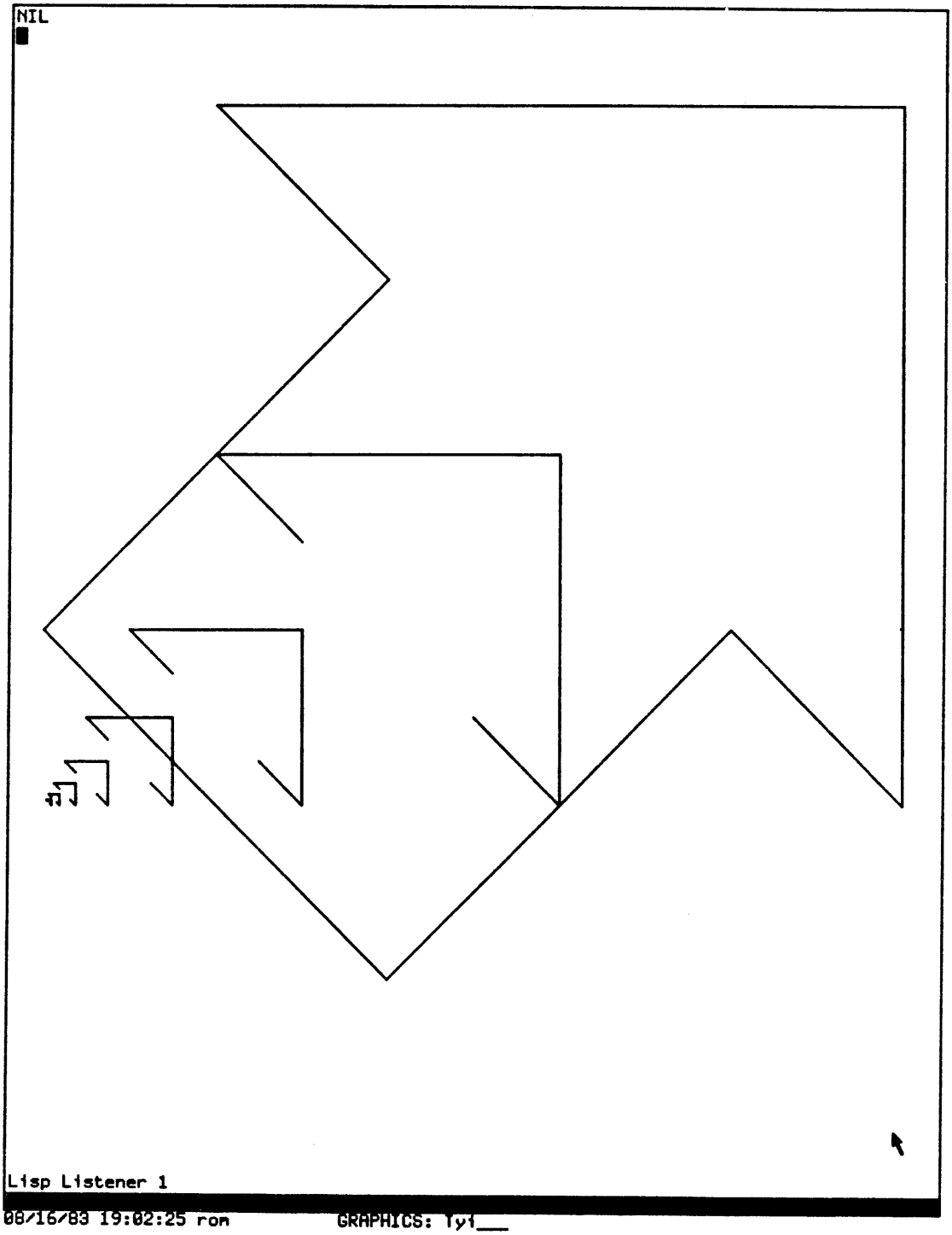



Figure 9. Output resulting from a faulty attempt to outline the small arrows recursively, with the second function call commented out.

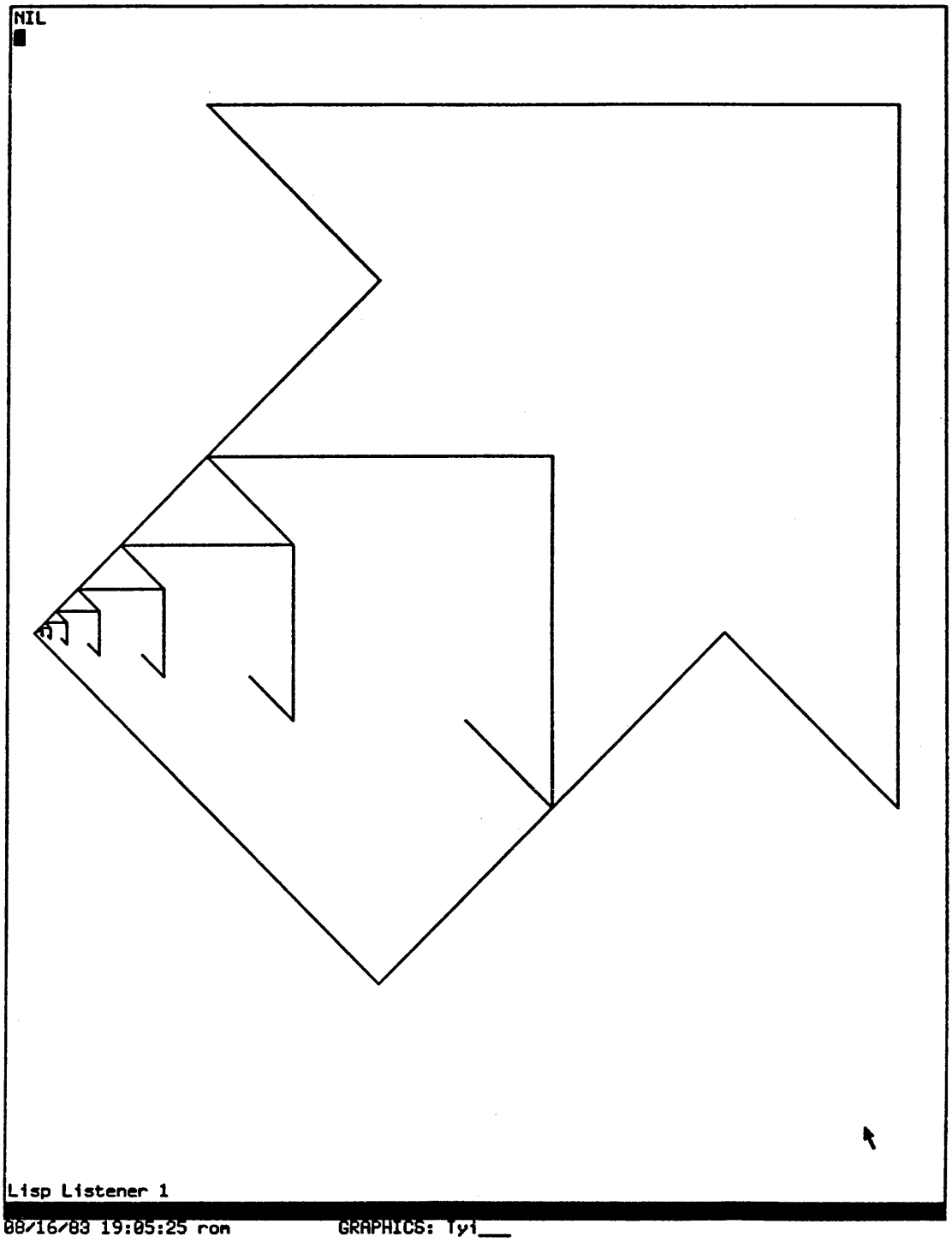



Figure 10. Output resulting from a corrected attempt to outline the small arrows recursively, with the second function call commented out.

## 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 (page 87) summarizes the correspondence between trace menu items and **trace** options. See the *Lisp Machine Manual*, section 26.3, page 457, for a description of the options.

**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 a 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 to **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 section 3.2.1, page 66).

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 breakpoint loop. Press HELP inside the Stepper or see the *Lisp Machine Manual*, section 26.5, page 464, for a list of commands. 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) |

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 (see section 4.4.2, page 86) |
| [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* |
| [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 |

| | |
|---|---|
| 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 in section 4.4.1 (page 84). 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 a 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 (see the *Lisp Machine Manual*, section 10.10, page 153, for more on encapsulations). 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-X). 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 (page 93). 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)
```

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*
```

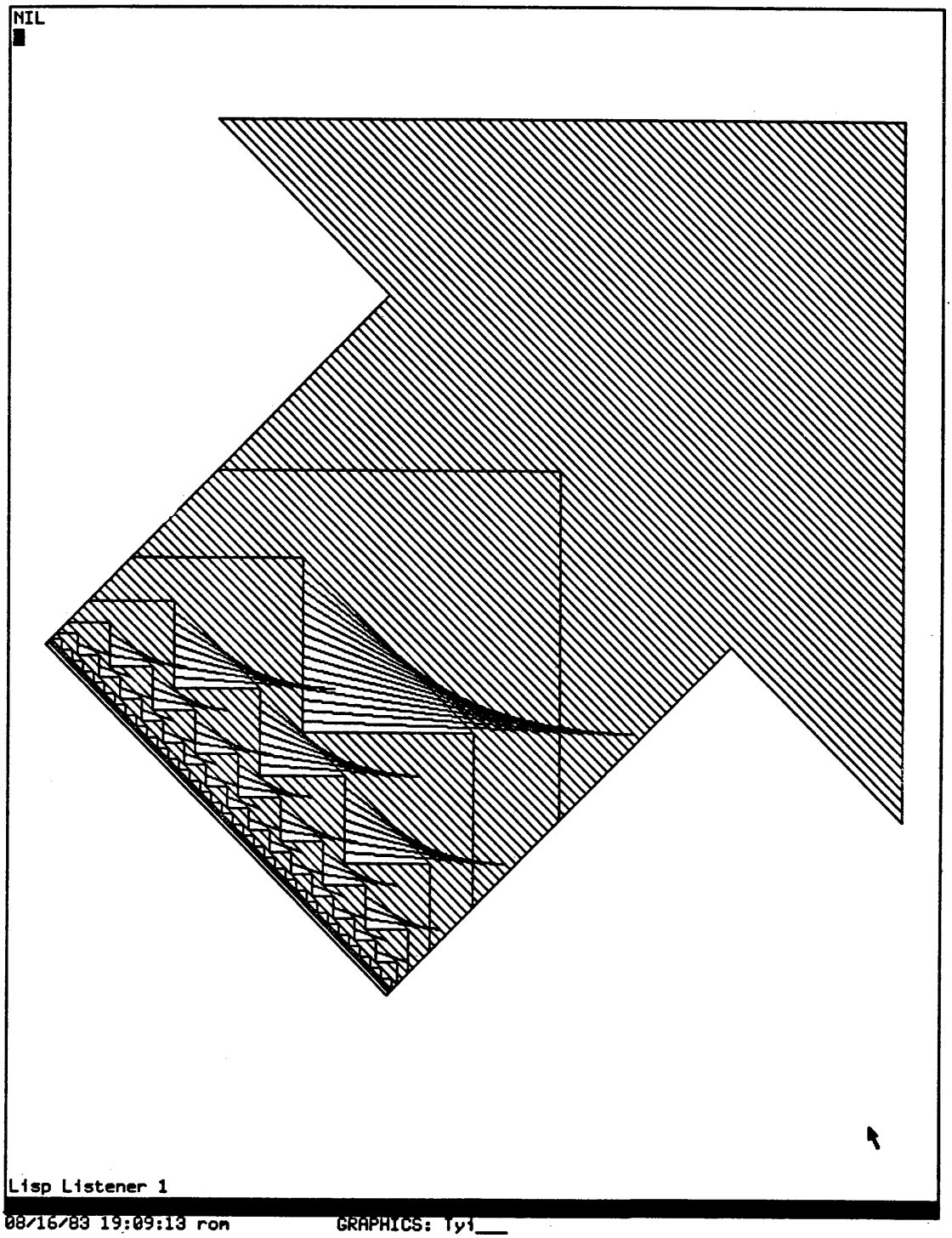



Figure 11. Output from the program with a bug in the function **draw-arrow-shaft-stripes**.

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-X) | 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 a 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].

See *Operating the Lisp Machine*, chapter 5, page 28, for more on the Inspector.

---

**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 (page 96) 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.

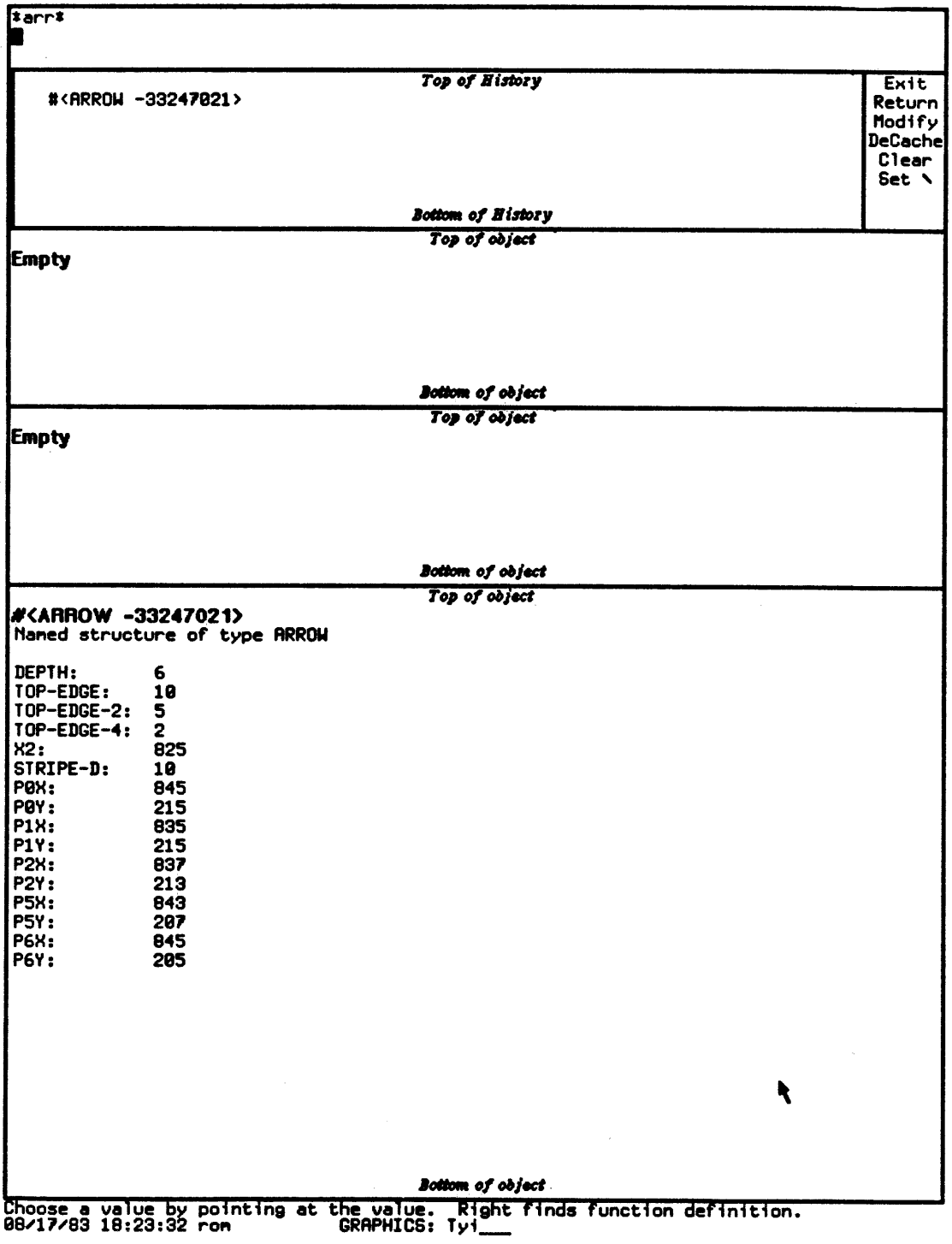



Figure 12. The Inspector window: inspecting an instance of a structure.

Figure 13 (page 98) 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 for 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 a 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 representation of the function's assembly-language instructions. |

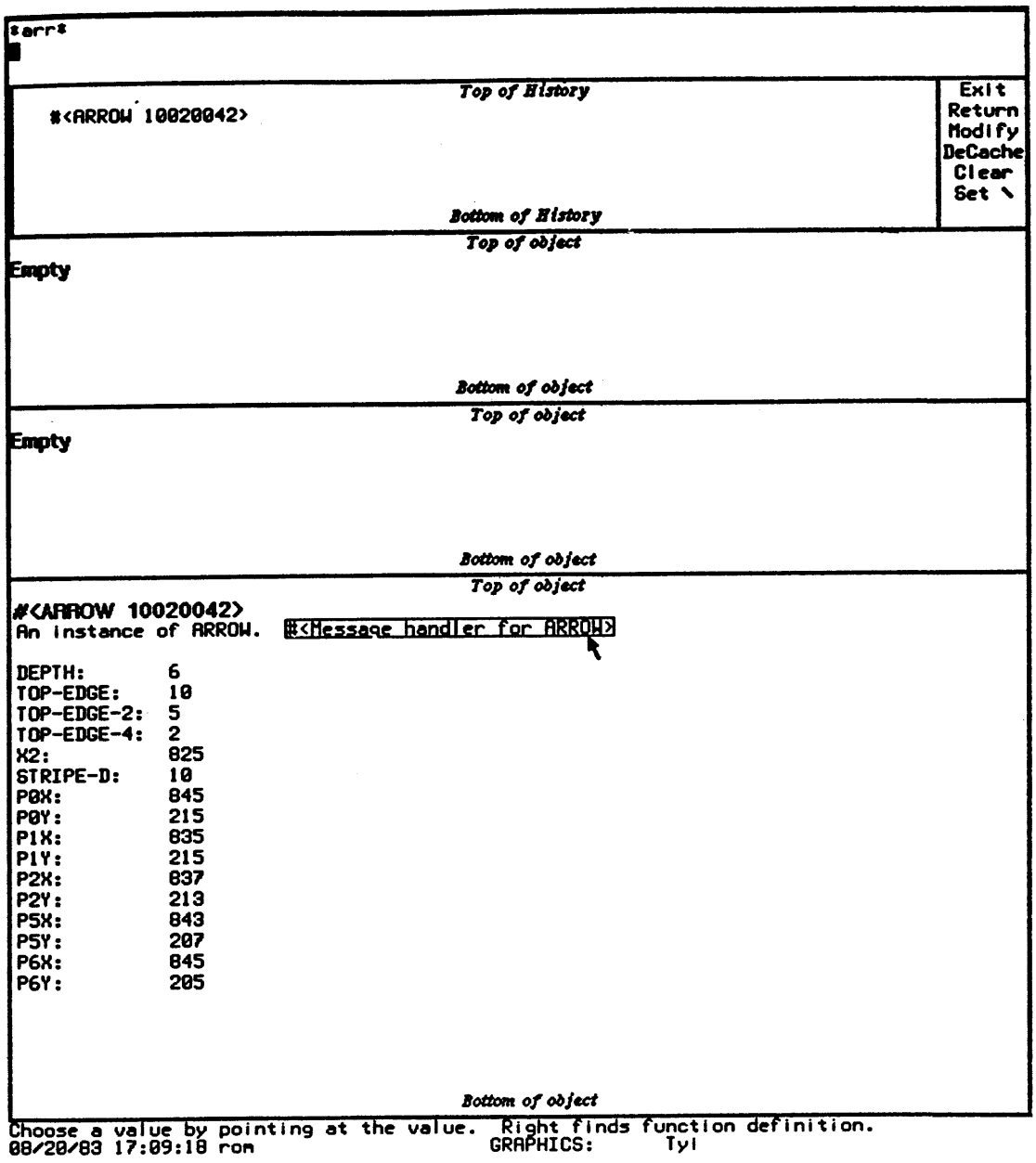



Figure 13. The Inspector window: inspecting an instance of a flavor.

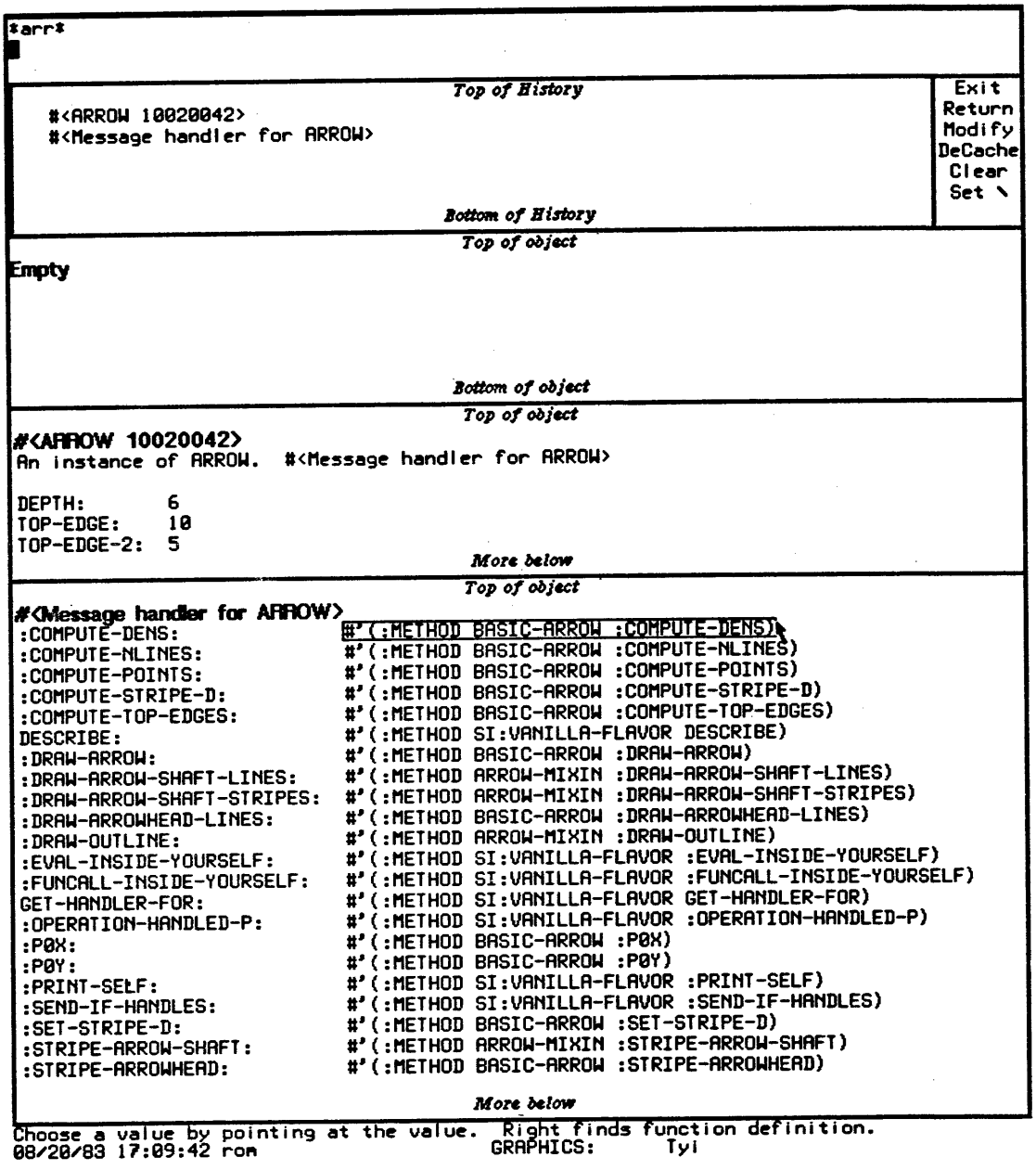



Figure 13, *continued.*

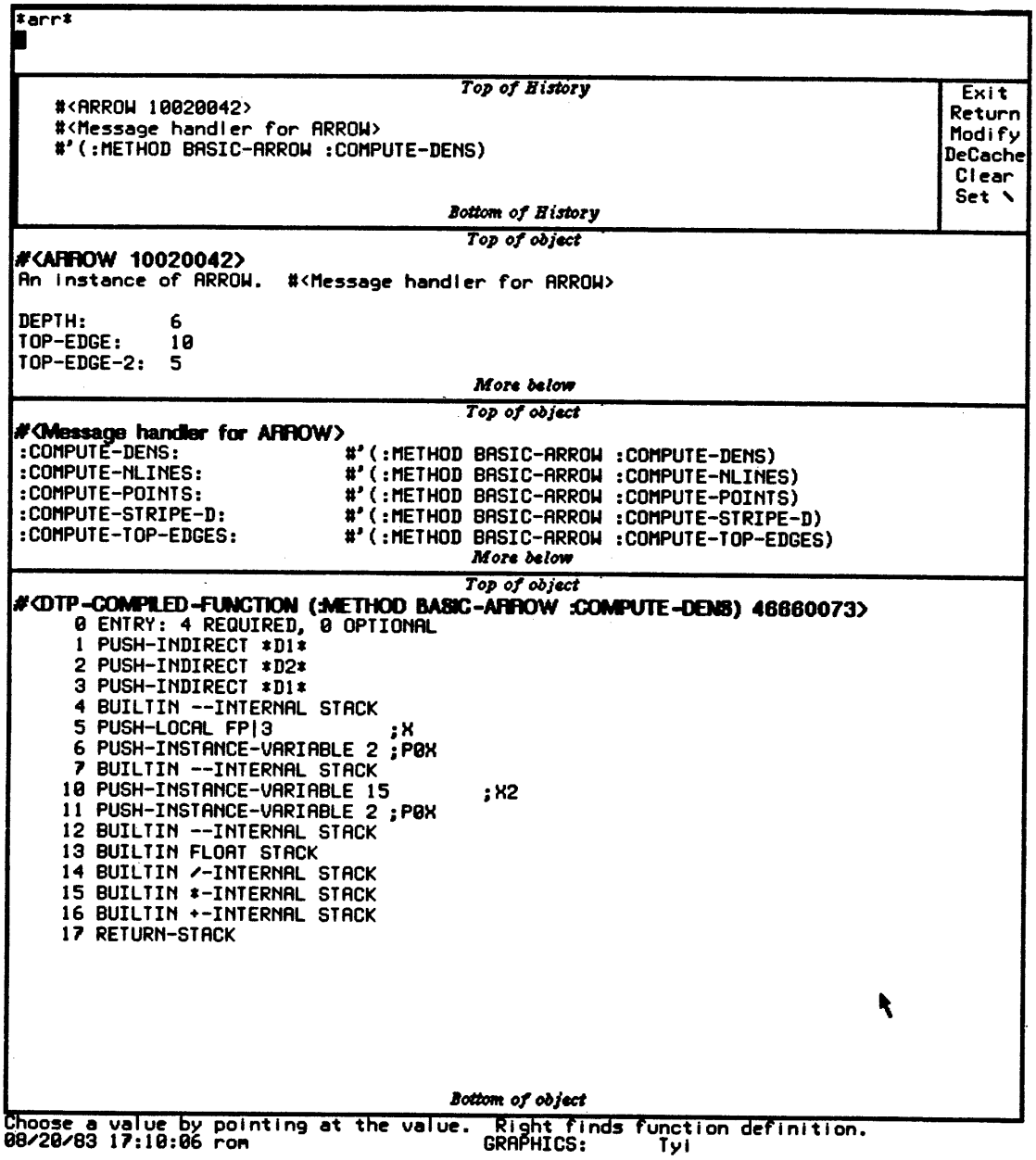



Figure 13, *concluded.*

# 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 documents:

- On flavors: *Lisp Machine Manual*, chapter 20, page 279
- On windows: *Introduction to Using the Window System*
- On menus: *Lisp Machine Choice Facilities*
- On conditions and errors: *Signalling and Handling Conditions*

## 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 *Lisp Machine Manual*, section 20.12, page 306.)

```
;;; 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:minimum-window**, 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:MINIMUM-WINDOW 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 *Lisp Machine Manual*, section 20.12, page 306.)
- 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 *System 210 Release Notes*, section 7.3, page 41.)

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 for **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 for **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 a 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 section 4.7, page 94). 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 *Introduction to Using the Window System*). 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. Section 5.2.2 (page 129) describes how to use this command to edit method definitions.

### 5.2.1 General Information

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 to **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 to **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 for **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.

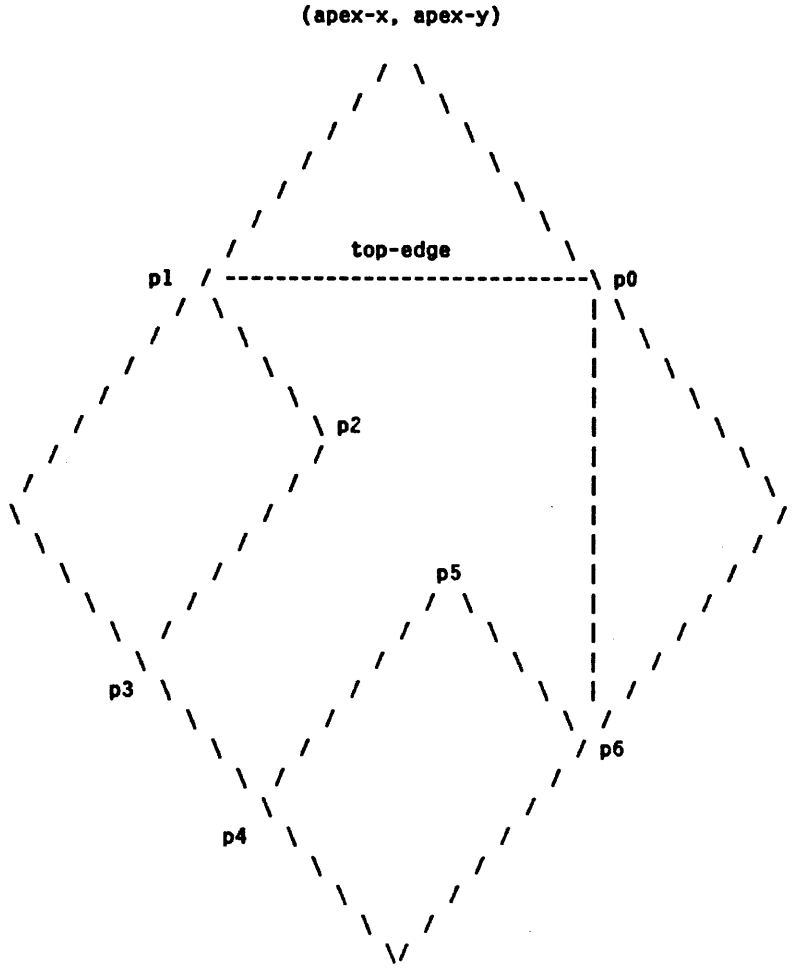
# Appendix A
# Calculation Module for the Sample Program

The program used as an example in this document draws the recursive arrow graphic on the document's cover. This appendix 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. Appendix B (page 147) contains the code for the flavors and methods that mediate between the program and the system output operations. Appendix C (page 165) contains a reproduction of the LGP graphic the program produces.

```
;;; -*- 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*.



```
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)))
```

# Appendix B
# 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 appendix contains Lisp code that defines the flavors and methods that mediate between the program and the system output operations. Appendix A (page 133) contains the code that calculates coordinates for the endpoints of the lines that compose the figure. Appendix C (page 165) contains a reproduction of the LGP graphic the program produces.

```
;;; -*- 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:MINIMUM-WINDOW 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)
```

# Appendix C
# 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 appendix contains a reproduction of the LGP graphic the program produces. Appendix A (page 133) contains Lisp code that calculates coordinates for the endpoints of the lines that compose the figure. Appendix B (page 147) contains the code that defines the flavors and methods that mediate between the program and the system output operations.

# Index