Program Development Tutorial

Tutorials of Lisp Programming in Genera

Introduction

Read this section when you are ready to start developing Lisp code in Genera. Our goal is to help you get started as quickly as possible, by developing a progression of examples which run the game of Life. The first example is very simple; it shows the mechanics of editing, compiling, and running a Lisp program in Genera. We then build on that example, by focusing on improving different aspects of it, such as adding a visually appealing display, adding a command-driven interface, and creating a "program framework" that gives it a user interface. Finally, we show how to use logical pathnames to make it easy and convenient for users to load and run your program.

Choose your own reading path through this material. If you are new to Genera, you will probably want to start at the beginning. If you are already familiar with interacting with the Lisp Listener and editing code, feel free to skip ahead to other subjects of interest. The sections are:

Getting Started Using Lisp in Genera
: For the Genera novice: how to use the Lisp Listener for reading, evaluating, and printing Lisp forms.

Developing a Simple Common Lisp Program: Life
: For the Genera novice: how to edit, compile, and run a simple Common Lisp program, the game of Life.

Easy Ways to Improve the Lisp Program
: This section shows how easy it is to improve a program by adding a graphic display, and by providing a command-driven interface.

Programming a User Interface for the Life Game
: Shows how to use the Symbolics paradigm for developing a user interface customized for your application, with a minimum of code. You can build your user interface by using a declarative language which insulates you from the low-level code that does the basic work of manipulating the screen.

Using Logical Pathnames for the Life Program
: Do you want to share your program with other users? To make it easy and convenient for network users to load and run your program, and for you to distribute it via tape to another site, you should use logical pathnames.

Getting Started Using Lisp in Genera

The Lisp Listener is the most basic tool for interacting with Lisp. Read this section to learn how to use the Lisp Listener for reading, evaluating, and printing Lisp forms.

Using the Read-Eval-Print Loop of the Lisp Listener

The Lisp Listener is the user interface for interacting with Lisp. It provides a read-eval-print loop. When you first boot a Symbolics machine, you are in the Lisp Listener. (At other times in the work session, you can select the Lisp Listener by pressing SELECT L.)

The prompt appears like this:

```
Command:
```

You can type in a Lisp form and it will be read, evaluated, and printed. For example:

```
Command:  (+ 3 4)
7
```

In the example above, the closing parenthesis ")" triggered the read-eval-print loop. There was no need to press RETURN.

The Symbolics documentation usually presents the interaction at a Lisp Listener without the prompt, and with => to indicate the returned value. Here are other examples of entering Lisp forms for evaluation:

```
(list 'A 'B) => (A B)
(setq seven 7) => 7
seven => 7
```

You can define a function and execute it using the Lisp Listener:

```
(defun fahrenheit-to-centigrade (degrees-c)
  (* (/ 5.0 9.0) (- degrees-f 32)))
=> FAHRENHEIT-TO-CENTIGRADE

(fahrenheit-to-centigrade 212) => 100
```

If you make a mistake in typing an expression to the system, you have two choices:

- Press ABORT or CLEAR INPUT and begin again.
- Edit your input.

You do not need to invoke the input editor explicitly. For a complete list of the commands available in the input editor, press c-HELP or see the section "Input Editor Commands".

Entering Commands and Lisp Forms to the Lisp Listener

You can use the Lisp Listener to enter commands to the Command Processor, and to enter forms to Lisp.

Commands have English names, and you enter one by typing the command and any arguments, and pressing RETURN. For example, you can give the HELP command to get a list of commands:

```
Command:  Help
```

By default, the Lisp Listener is in *command-preferred mode*. This means that if you enter something that starts with an alphabetic character, it is treated as a command for the Command Processor. If there is no command with that name, and there is a Lisp symbol with that name, then the input is treated as a Lisp form.

A Lisp form that starts with an open parenthesis is always treated as a Lisp form, since the "(" is not alphabetic.

In command-preferred mode, if you want to get the value of a symbol whose name starts with an alphabetic character, you should precede the symbol with a comma "," to indicate that it is intended for the Lisp read-eval-print loop. The comma indicates that the input is a Lisp form even though it starts with an alphabetic character.

For example, if you enter Help followed by RETURN, the Help command is executed. You might have a symbol named Help:

```
(setq help "sos") => "sos"
```

You can get the value of the symbol named Help as follows:

```
,help => "sos"
```

For more information on command-preferred mode and the other modes: See the section "Setting the Command Processor Mode".

For detailed information on entering commands: See the section "Entering Commands".

Short-Cut Ways to Enter Lisp Forms

Using Variables that Provide Short-Cuts

Certain variables provide short-cut ways to enter Lisp forms to a Lisp Listener. For example, the variable * holds the result of the previous evaluation.

After filling up your car with gasoline, you might want to calculate the miles driven on the last tank, and use the result of that to calculate the gas mileage. You can do this in two steps as follows:

```
(- 32841 32609) => 232
(/ * 7.5) => 30.933332
```

Above, the form **(/ * 7.5)** is the same as **(/ 232 7.5)**, because 232 is the result of the previous evaluation.

These variables are particularly useful for typing Lisp interactively, especially for examining your state in the Debugger.

-	Holds the form being evaluated.
+	Holds the previously evaluated form.
++	Holds the form evaluated two interactions ago.
+++	Holds the form evaluated three interactions ago.
*	Holds the result of the previous evaluation.
**	Holds the result of the form evaluated two interactions ago.
***	Holds the result of the form evaluated three interactions ago.
/	Holds a list of the results of the previous evaluation.
//	Holds a list of results from two interactions ago.
///	Holds a list of the results from three interactions ago.

Using the Mouse

You can enter Lisp forms by using the mouse. If you entered a Lisp form and want to evaluate it again, you can position the mouse over that form and click Left to enter the form to the Lisp Listener.

Using the Command History

A Lisp Listener keeps track of the *command history*, which includes Lisp forms entered, as well as commands entered. By yanking a Lisp form or command, you can easily execute it again, or edit it and then execute the edited version.

`c-m-Y` yanks the last element of the history. `m-Y` yanks the next previous element. Thus you can press `c-m-Y` followed by `m-Y m-Y ...` to yank successively further back elements in your input history. `c-m-0 c-m-Y` lists the elements of the history. A numeric argument to `c-m-Y` yanks the element of the history specified by the argument. For more information, see the section "Command History".

Developing a Simple Common Lisp Program: Life

Read this section to learn how to edit, compile, and run a simple Common Lisp program, the game of Life.

The Game of Life

We will develop a program that plays the game of "Life." Life simulates a community of beings called cells. The rules of Life indicate whether a given cell will live or die in the next generation, depending on its environment. If the cell is too crowded by its neighbors, or too isolated from other cells, it dies. Otherwise the environment is deemed acceptable, and the cell lives. Specifically:

- If an empty cell has exactly 3 live neighbors, a cell is born there.
- If an empty cell has any other number of live neighbors, no cell is born.
- If a live cell has 2 or 3 live neighbors, it stays alive.
- If a live cell has any other number of live neighbors, it dies.

Some implementations of Life have a finite but unbounded domain, by treating a cell that lives on the edge of the domain as a neighbor of the corresponding cell on the other edge of the domain. For the purpose of this example we have a simpler version of Life, characterized by the following statements:

- Cells live in a two-dimensional array.
- A cell typically has eight neighbors, those adjacent to it.
- Cells on the border of the array have less than eight neighbors.

The Life Program in Common Lisp

This section contains the Common Lisp code that implements the Life program. This code is available in the file SYS:EXAMPLES;COMMON-LISP-LIFE.LISP.

```
;;; Both *game-board* and *next-game-board* are 2-d arrays that store
;;; the state of the Life game board.  They both have an outer border
;;; composed of dead cells, which makes it convenient to compute the
;;; number of live neighbors.  The inner part contains cells that are
;;; initialized randomly dead or alive.  A live cell has the value 1; a
;;; dead cell has the value 0.

;;; This variable controls the size of the Life game board.
(defvar *number-cells-on-axis*  30
  "Number of cells on each axis of game board")

(defun make-game-board ()
  (make-array (list (+ 2 *number-cells-on-axis*)
                    (+ 2 *number-cells-on-axis*))
              :initial-element 0))              ; all dead cells to start

;;; *game-board* stores the current state of the Life game board
(defvar *game-board* (make-game-board))

;;; *next-game-board* stores the upcoming state of the Life game board
(defvar *next-game-board* (make-game-board))
```

```
(defmacro do-over-board ((x y) &body body)
  `(do ((,y 1 (+ 1 ,y)))
       ((= ,y *number-cells-on-axis*))
     (do ((,x 1 (+ 1 ,x)))
         ((= ,x *number-cells-on-axis*))
       ,@body)))

(defun initialize-board-with-cells (board)
  "Initialize inner part of the array with cells.
Cells are randomly chosen to be alive or dead."
  (do-over-board (x y)
    (setf (aref board x y)
          (random 2))))

(defun display-game-board ()
  (terpri)
  (do-over-board (x y)
    (when (= x 1) (terpri))
    (let ((cell-status (aref *game-board* x y)))
      (cond ((= 0 cell-status) (format t " "))
            ((= 1 cell-status) (format t "X"))
            (t (error "Unrecognized cell status."))))))

(defun play-life-game (&optional (generations 3))
  (initialize-board-with-cells *game-board*)
  (display-game-board)                          ;display 0th generation
  (step-generation generations))

(defun step-generation (&optional (generations 1))
  (do ((i 0 (+ i 1)))
      ((= i generations))
    (calculate-next-board)
    (display-game-board)))
```

```
(defun calculate-next-board ()
  (do-over-board (x y)
    ;; For each cell, count the number of live neighbors, and apply
    ;; the Life rules to see whether cell should live or die in the
    ;; next generation.
    (let* ((live-neighbors
             (+ (aref *game-board* x (1- y))
                (aref *game-board* x (1+ y))
                (aref *game-board* (1- x) y)
                (aref *game-board* (1+ x) y)
                (aref *game-board* (1- x) (1- y))
                (aref *game-board* (1- x) (1+ y))
                (aref *game-board* (1+ x) (1- y))
                (aref *game-board* (1+ x) (1+ y))))
           (next-status
             (cond ((= 0 (aref *game-board* x y))       ;dead cell
                    (if (= live-neighbors 3) 1 0))
                   (t                           ;live cell
                    (if (or (= live-neighbors 2)
                            (= live-neighbors 3))
                        1 0)))))
      (setf (aref *next-game-board* x y) next-status)))
  ;; Switch next and current boards
  (rotatef *game-board* *next-game-board*))
```

Developing Life in the Editor

This section goes quickly through the steps of developing the Life game in the Zmacs editor.

1. Enter the Editor by pressing `SELECT E`.

2. Create a new file to store the Life program.

   Give the Zmacs Find File command by pressing `c-X c-F`. You are prompted for a filename. Name the file something like `life.lisp`.

   The `.lisp` extension tells Zmacs that this is a Lisp file, and automatically puts your editor buffer into Lisp Mode, which enables you to use the Zmacs commands that assist in developing Lisp programs. See the section "Zmacs Major and Minor Modes".

3. Set the attributes of the buffer and file. Give the command `m-X` Set Lisp Syntax. At the prompt, enter Common-Lisp. Answer Yes to the question about entering this in the file attribute list. You will notice that the first line in the buffer is:

```
;;; -*- Syntax: Common-Lisp -*-
```

This is a *file attribute list*, which contains information about the file; in this case, it states that the Lisp syntax of the file is Common-Lisp, not Zetalisp. The file attribute list can contain other information, such as the package and base of the file. For more information, see the section "Buffer and File Attributes in Zmacs".

We want our program to be in the **cl-user** package. To do this, give the command Set Package (m-X). At the prompt, enter **cl-user**. Answer Yes to the question about entering this in the file attribute list.

See the section "Creating a File Attribute List".

Also, it is useful to add a file attribute indicating that this buffer is in Lisp Mode. To do this, give the command Lisp Mode and answer Yes to the question about entering this in the file attribute list.

4. Put the Life program into the editor buffer and file.

   If you want to practice using Zmacs, you can use the editor to enter the program. For introductory information on Zmacs: See the section "Using the Zmacs Editor". For a Zmacs tutorial on using Zmacs: See the section "Workbook: Zmacs".

   Alternatively, you can use the Insert File (m-X) command to copy the contents of the file SYS:EXAMPLES;COMMON-LISP-LIFE.LISP into the buffer and save the file.

5. Compile the buffer.

   Give the command Compile Buffer (m-X).

   If the compiler gives any error messages, fix the problem and compile the buffer again, until there are no errors. There is a shortcut to compiling the whole buffer; when you fix a bug in a function, you can use c-sh-C with the cursor in the Lisp definition to compile just that definition.

6. Select the Lisp Listener by pressing SELECT L.

7. Run the program by calling the **play-life-game** function. To see one generation after the initial display, enter:

   ```
   (play-life-game 1)
   ```

   To watch this configuration go through more generations, call the **step-generation** function:

   (step-generation *number-of-generations*)

Bread-and-Butter Techniques for Editing Lisp Code

This section describes the most useful techniques for developing code in the editor: commands that locate source code automatically, enabling you to see the arguments to a function, edit a function, find the callers of a function, and compile code incrementally.

Editing a definition: m-.

Probably the most useful Zmacs tool is the m-. command. Whenever you want to start editing a definition, there is no need to worry about what file it is in, or use search commands to find the definition within a buffer. Instead, simply use the m-. command.

For example, assume you are working on the **step-generation** function:

```
(defun step-generation (&optional (generations 1))
  (do ((i 0 (+ i 1)))
      ((= i generations))
    (calculate-next-board)
    (display-game-board)))
```

You might want to look at the definition of **display-game-board**, which is called in **step-generation**. To do so, give the m-. command and enter the name of the function, **display-game-board**. A short-cut is to point the mouse on the symbol **display-game-board** and press m-LEFT.

Zmacs locates the definition of **display-game-board** and places your cursor at that definition. In this example, the definition happens to be in the same file. If the function were defined in a different file, Zmacs would have read the file into a buffer, made that buffer current, and placed your cursor at its definition.

Showing the arguments of a function: c-sh-A

Very often when you are developing a program, you want to call a function but don't know offhand the arguments that the function takes. For example, you might stop at this point:

```
(defun make-game-board ()
  (make-array
```

If you forget the arguments to **make-array**, position your cursor on **make-array** and press c-sh-A. Zmacs responds by displaying the arguments to **make-array** in the type-out window.

Using these commands in other contexts

The Edit Definition and Show Arglist commands are available in contexts other than Zmacs. For example:

- When calling a function in the Lisp Listener or Debbuger, and you can't recall its arguments, press c-sh-A.

- When a function definition is presented on the screen (the name or the function spec is visible) in the Lisp Listener or Debugger, you can click Left to edit the definition. You can also click Right for a menu of operations on the definition.

Updating a function's callers: List Callers and Edit Callers

Suppose that you defined a function **print-game-board** but decided later to rename it **display-game-board**. Such a change would affect all callers of the function, so you need to locate and edit the definitions of all callers.

To get an idea of how many functions call **print-game-board**, position the cursor on **print-game-board** and use `m-X` List Callers. Zmacs responds by listing the callers in the type-out window.

When you are ready to edit the callers, you can edit them one by one by entering `c-.` Zmacs brings you to the definition of the first caller so you can edit it. This might entail moving the cursor within the current buffer, or switching to another buffer, or reading in the file where the caller is defined. If there are additional callers, you can cycle through them by pressing `c-.`

The command `m-X` Edit Callers is another way to do the same thing. It does not list the callers, but simply brings you to the definition of the first caller and lets you cycle through the others by pressing `c-.`

Using incremental compilation: `c-sh-C`

To continue the example of changing the name of a function and then editing its callers, you need to compile the callers to make the change take effect. It is not necessary in Genera to compile the entire program; instead, compile only the definitions that have changed. There are several ways to do this:

- Each time you edit a function, compile the new definition immediately by pressing `c-sh-C` with the cursor on the definition. The definition is compiled immediately.

- Edit all the affected functions, and use one of these commands:

  Compile Changed Definitions of Buffer
  Compile Changed Definitions

  The first command compiles only the changed definitions within a single buffer, and the second command compiles changed definitions in all buffers.

Easy Ways to Improve the Lisp Program

You'll see how easy it is to improve a program by adding a graphic display, and by providing a command-driven interface.

When you are ready to do more work on the user interface of Life, see the section "Programming a User Interface for the Life Game".

Adding a Graphic Display to the Life Game

It's tempting to make the display of the Life game more visually appealing. It is an easy matter to add a graphic display; we rewrite **display-game-board** to use **graphics:with-room-for-graphics** and **graphics:draw-rectangle** as follows:

```
(defun display-game-board ()
  (graphics:with-room-for-graphics
    (t (+ 10 (* 5 *number-cells-on-axis*)))
    (do-over-board (x y)
      (let ((cell-status (aref *game-board* x y)))
        (cond ((= 0 cell-status))
              ((= 1 cell-status)
               (let ((xdraw (* x 5)) (ydraw (* y 5)))
                 (graphics:draw-rectangle
                   xdraw ydraw (+ xdraw 5) (+ ydraw 5))))
              (t (error "Unrecognized cell status."))))))
```

This function is in the file SYS:EXAMPLES;COMMON-LISP-LIFE-WITH-GRAPHICS.LISP.

We compile the new definition by pressing `c-sh-C`. The next time we play the Life game, the gameboard appears as in Figure !.

Figure 84. Graphic Display of the Life Game

For simple graphics output, **graphics:with-room-for-graphics** is the macro to use. It binds the local environment to establish a Cartesian coordinate system for doing graphics output. The origin <0, 0> of the local coordinate system is in the lower left corner of the area created.

graphics:with-room-for-graphics takes two optional arguments, *stream* and *height*. We supplied **t** for the stream argument; this indicates that output should be sent to the default stream, ***standard-output***. The second argument is the height, which is the number of pixels needed in the vertical direction. We requested the height to be five times the number of cells on an axis.

Remember that a Dynamic Lisp Listener saves the history of your interaction, so the visible region of the window is usually just a portion of the entire window. (You can scroll the window to explore the regions of the window that are currently concealed.) When doing simple graphics, it is convenient to deal with a small, local coordinate system rather than to deal with the coordinate system used by the Dynamic Lisp Listener for its entire history. The **graphics:with-room-for-graphics** macro provides such a coordinate system.

For more information on performing graphic output, see the section "Basic Graphic Output Facilities".

In this simple case, we used only one drawing function, **graphics:draw-rectangle**. For information on other drawing functions, see the section "Drawing Functions".

Adding a Command Interface to the Life Game

Currently, users play the Life game by entering a Lisp form to the Dynamic Lisp Listener:

```
(play-life-game 3)
```

We can add a command interface to the game, so users can play it by giving a command:

```
Play Life 3
```

Benefits of a Command-driven Interface

The advantages of a command-driven interface are:

- Commands are usually perceived as a more desirable user interface than are Lisp forms, because commands more closely resemble a natural language; in this case, English.

- Completion is supported. The user can enter the command in an abbreviated format to save typing. For example, users can type the following abbreviation for the command, and the command is recognized and executed:

  ```
  P L 3
  ```

- Prompting is supported. Once the command name has been entered, the Command Processor prompts for the arguments. For example, when the user has entered Play Life, the CP prompts for the argument "Number of generations", and also displays the default value for that argument:

```
Play Life (Number of generations (default 3)):
```

- Online help is available. If you need more information about an argument expected, you can press HELP. For example, if you press HELP in response to the prompt for "Number of generations" above, the result is the following additional information:

```
You are being asked to enter an integer greater than
or equal to 1
```

How to Define a Command

Use **cp:define-command** to define a new command. For example, we can define the Play Life command as follows:

```
(cp:define-command (com-play-life :command-table "USER")
    ((generations '((integer 1 *))
                  :prompt "Number of generations"
                  :default 3))
   (play-life-game generations))
```

This function is in the file SYS:EXAMPLES;COMMON-LISP-LIFE-WITH-COMMANDS.LISP.

Here is a brief summary of **cp:define-command** syntax:

cp:define-command *name-and-options args* &body *body*
: This defines a CP command named *name* that reads in from the user a set of values bound to variables specified in *args*. When the command is invoked, it executes the *body*.

Our command has only one argument, named *generations*. It must be an integer greater than or equal to 1. We used the **:prompt** option to give the prompt "Number of generations", and we used the **:default** option to specify that if the user does not specify a value for *generations*, the default will be 3.

The Presentation Type of an Argument

When defining a command that takes arguments, you indicate the type of argument expected. The Play Life command, for example, expects its *generations* argument to be an integer that is greater than or equal to 1. In this example, the form **(integer 1 *)** is used to express the type of argument expected. This is one example of a *presentation type*.

By expressing the expected type of an argument as a presentation type, you gain the following advantages:

- The argument types are checked when they are read. The Command Processor enforces the requirement expressed by the presentation type; it rejects an argument that is not of the expected type. Users can correct typing mistakes by pressing RUBOUT and editing the argument given; this is not an issue in the simple command shown here, but it can save time when entering more complicated commands.

- Users can press the HELP key to get information about the type of argument expected.

- If users want to enter as input some value that appears on the screen (in another context, in some part of the history of the Lisp Listener), they can click on the displayed value and it will be entered as input to the command. In other words, when a given type of argument is expected, any pieces of data displayed on the screen that are of that type are made mouse-sensitive.

The ability to click on a piece of data appearing on the screen can be very useful, although for an integer, it is usually easier to type it in than to seek for the integer somewhere on the and click on it. Consider other kinds of data, such as pathnames. When a directory listing is visible on the screen, you can operate on a given pathname by giving a command such as Delete File, and clicking Left on a pathname.

Programming a User Interface for the Life Game

You can define a new context for the Life game to run in, including:

- A SELECT key for selecting the Life game.

- A window customized for Life.

- A command table customized for Life, which can include all the Life commands and some generally useful commands.

- The ability to run separate Life games.

The Symbolics paradigm for programming a user interface enables you to develop a user interface customized for your application, with a minimum of code. You can build your user interface by using a declarative language which insulates you from the low-level code that does the basic work of manipulating the screen. The backbone of the declarative language is **dw:define-program-framework**.

The code in this section is in the file SYS:EXAMPLES;COMMON-LISP-LIFE-WITH-PROGRAM-FRAMEWORK.

Program Framework for the Life Game

In this section, we modify the Life program to use a program framework.

The first step is to use **dw:define-program-framework**:

```
(dw:define-program-framework life
  :select-key #\square
  :command-definer t
  :command-table
     (:inherit-from '("colon full command"
                      "standard arguments"
                      "input editor compatibility")
      :kbd-accelerator-p 'nil)
  :state-variables ((number-cells-on-axis 30)
                    (game-board (make-game-board 30))
                    (next-game-board (make-game-board 30)))
  :panes ((title :title
            :default-character-style '(:swiss :bold :large)
            :redisplay-string "Life")
          (display :display
            :more-p nil)
          (listener :listener))
  :configurations
     '((main
         (:layout
           (main :column title display listener))
         (:sizes
           (main (title 1 :lines) (listener 10 :lines)
                 :then (display :even))))))
```

The **dw:define-program-framework** form above has the following effects:

- Enables you to select the Life game by using SELECT SQUARE.
- Defines a macro **define-life-command**, which you can use to define commands for the Life game. The syntax is much like that of **cp:define-command**, but the Life commands are automatically installed in Life's command table.
- Causes Life's command table to inherit from three command tables that are generally useful.
- Defines three *state variables*, which will replace three global variables used in previous versions of Life.
- Defines three panes of the Life window: a title pane; a display pane, where the Life gameboard will be displayed; and a listener pane, which is used for entering Life commands.
- Specifies the layout and size of the panes.

For reference information on these options, see the function **dw:define-program-framework**.

We now replace the Play Life command with two commands, called Initialize and Step:

```
;;; note that dw:define-program-framework defined the
;;; macro define-life-command

(define-life-command (com-step)
    ((generations '((integer 1 *))
                  :prompt "Number of generations"
                  :default 1))
   (step-generation self generations))

(define-life-command (com-initialize)
    ()
   (initialize-board-with-cells self game-board)
   (display-game-board self))
```

We need to modify portions of the rest of the Life program to make it work in the program framework. We make and discuss these modifications elsewhere; see the section "Using Flavors in the Life Program Framework".

Using Flavors in the Life Program Framework

One advantage of using a program framework is that it lets you run different instantiations of your program on the same machine. Instead of having one gameboard, there is one gameboard per Life activity. Instead of having one display, there is one display per Life activity.

You can use this expanded functionality very easily. To set up your first Life activity, press SELECT SQUARE. To set up other Life activities, press SELECT c-SQUARE. You can cycle among the various Life activities by pressing SELECT SQUARE.

The separation of Life activities is accomplished by using Flavors. The **dw:define-program-framework** macro defines a flavor named **life** to represent the Life program framework. Each Life activity is represented by an instance of this flavor. This change requires us to make several changes in our program.

The purpose of each of these changes is the same, namely to change the program from a global scope (where state is stored in global variables and output goes to ***standard-output***) to a local scope (where state is stored in instance variables, and output goes to the appropriate place).

The changes are:

- Several global variables need to be changed to *state variables*, which are instance variables of the **life** flavor. The **:state-variables** option to **dw:define-program-framework** defined three state variables, which replace ***number-cells-on-axis***, ***game-board***, and ***next-game-board***.

```
:state-variables ((number-cells-on-axis 30)
                  (game-board (make-game-board 30))
                  (next-game-board (make-game-board 30)))
```

- Several functions need to be converted into methods that specialize on the flavor **life**. These methods can use the state variables in the places where the old functions used global variables. (Note that you can also use the state variables in the bodies and argument lists of **define-life-command**.)

- The macro **do-over-board** needs to access the state variables of **life** rather than the old global variables. This means that we must take care to call the macro only within methods for the **life** flavor; in any other context, the macro would not have access to the state variables. This is acceptable, since we intend to use **do-over-board** only within this context.

- We need to direct the output of Life's display to the appropriate pane, instead of to ***standard-output***. This change occurs in the function **display-life-game.** The following code fragment of that function shows how to get the display-pane:

  ```
  (let ((pane (send dw:*program-frame* :get-pane 'display)))
    ...)
  ```

  The variable **dw:*program-frame*** is bound to the current instance of the window to which **life** is connected.

Now we show the modifications to the Life program that make it work in the program framework:

```
;;; modify so that number-cells-on-axis becomes an argument
;;; instead of the old global variable
;;;
(defun make-game-board (number-cells-on-axis)
  (make-array (list (+ 2 number-cells-on-axis)
                    (+ 2 number-cells-on-axis))
              :initial-element 0))      ; cells dead at start

;;; now a method
;;;
(defmethod (step-generation life) (&optional (generations 1))
  (do ((i 0 (+ i 1)))
      ((= i generations))
    (calculate-next-board self)
    (display-game-board self)))
```

```
;;; just like previous version, except for changing global variable
;;; *number-cells-on-axis* to state variable number-cells-on-axis
;;; Note that this macro can be used only in methods of life!
;;;
(defmacro do-over-board ((x y) &body body)
  `(do ((,y 1 (+ 1 ,y)))
       ((= ,y number-cells-on-axis))
     (do ((,x 1 (+ 1 ,x)))
         ((= ,x number-cells-on-axis))
       ,@body)))

;;; now a method
;;;
(defmethod (initialize-board-with-cells life) (board)
  (do-over-board (x y)
    (setf (aref board x y) (random 2))))

;;; now a method,
;;; also changed output from the destination *standard-output*
;;; to the destination of the display-pane
;;;
(defmethod (display-game-board life) ()
  (let ((pane (send dw:*program-frame* :get-pane 'display)))
    (graphics:with-room-for-graphics
       (pane (+ 10 (* 5 number-cells-on-axis)))
      (do-over-board (x y)
        (let ((cell-status (aref game-board x y)))
          (cond ((= 0 cell-status))
                ((= 1 cell-status)
                 (let ((xdraw (* x 5)) (ydraw (* y 5)))
                   (graphics:draw-rectangle
                     xdraw ydraw (+ xdraw 5) (+ ydraw 5)
                     :stream pane)))
                (t (error "Unrecognized cell status."))))))))
```

```
;;; just like previous version, except that it is a method,
;;; and the names of global variables are changed to the
;;; state variables
;;;
(defmethod (calculate-next-board life) ()
  (do-over-board (x y)
    ;; For each cell, count the number of live neighbors, and apply
    ;; the Life rules to see whether cell should live or die in the
    ;; next generation.
    (let* ((live-neighbors
             (+ (aref game-board x (1- y))
                (aref game-board x (1+ y))
                (aref game-board (1- x) y)
                (aref game-board (1+ x) y)
                (aref game-board (1- x) (1- y))
                (aref game-board (1- x) (1+ y))
                (aref game-board (1+ x) (1- y))
                (aref game-board (1+ x) (1+ y))))
           (next-status
             (cond ((= 0 (aref game-board x y))         ;dead cell
                    (if (= live-neighbors 3) 1 0))
                   (t                           ;live cell
                    (if (or (= live-neighbors 2)
                            (= live-neighbors 3))
                        1 0)))))
      (setf (aref next-game-board x y) next-status)))
  ;; Switch next and current boards
  (rotatef game-board next-game-board))
```

Using Logical Pathnames for the Life Program

Do you want to share your program with other users? To make it easy and convenient for network users to load and run your program, and for you to distribute it via tape to another site, you should use logical pathnames.

Benefits

A logical pathname creates a valuable abstraction which separates the way users access files and the actual place where the files are stored. Thus:

- When users access files with logical pathnames, you can store the files wherever you like, and in fact move them at will, without needing to inform users that the files have moved.

- You can make a tape of the files, bring it to another site, and restore the files to any physical machine at the site.

What are Logical Pathnames?

A logical pathname is a kind of pathname that doesn't correspond to any particular physical host. Instead, a "logical host" is defined, and a translation table maps the logical host to an actual physical machine.

One important use of logical pathnames is the "SYS" host, a logical host used to store the Genera system software files. Symbolics distributes Genera software by naming all files with logical pathnames whose host component is the host named SYS. Each site has a translation table mapping the SYS host to one or more physical machines at the site.

Example of Using Logical Pathnames

To use logical pathnames, you need to set up a *translations file*; this is a file named SYS:SITE;*logical-host-name*.TRANSLATIONS. That file should contain a call to **fs:set-logical-pathname-host**, which defines a logical host and its logical directories.

For example, you might create a file named SYS:SITE;GAMES.TRANSLATIONS, which contains the following form:

```
(fs:set-logical-pathname-host "GAMES" :physical-host "Stony"
          :translations '(("life;" ">examples>life>")
                          ("test-suite;" ">examples>test-suite>")))
```

The form defines a logical host named GAMES, which corresponds to the physical host named STONY. There are two logical directories:

```
games:life;
games:test-suite;
```

If you store your executable program in a file STONY:>EXAMPLES>LIFE>COMMON-LISP-LIFE.BIN, your users can load it by giving the command:

```
Load File GAMES:life;common-lisp-life
```

If you decide to store the files on a different machine, or in different directories, you can simply edit the translations file to reflect the new physical locations of the files. Users who access the games files after you have made the change will find them by using the same logical pathnames, but the system will find the files in their new location. However, users who have already accessed files via these logical pathnames in their work session will have to reload the translations file before accessing more of the games files.

To distribute these files to another site, you can make a distribution tape containing the contents of the logical directories GAMES:LIFE; and GAMES;TEST-SUITE;. At the other site, you will need to create a translations file that specifies a physical host at that site, and then restore the directories.

References to Related Information

The example here makes very basic use of logical pathnames, which is often all that is needed. However, when a program consists of many directories and files, you might need to use additional features of logical pathnames, such as:

- To use wildcards in the translations to match more than one directory name, see the section "Wildcard Matching in Logical Pathnames".
- To store some of the files on one host, and other files on another host, see the section "Logical Translations to Multiple Physical Hosts".
- To store files on VAX/VMS machines, see the section "Logical Pathname Translation for VAX/VMS Hosts".

For information on how Symbolics uses logical pathnames to distribute Genera software, see the section "Logical Pathnames".

For information on how logical pathnames fit into the pathname system, see the section "Pathnames".

Basic Tools for the Lisp Programmer

This section introduces new users to the time-saving tools in the editor and the Lisp Listener.

Editing Lisp Code

Programmers who are interested in learning the Zmacs tools for editing Lisp code will benefit by reading this section.

For those who prefer to learn an editor by using it instead of reading about it, there is one important and easy lesson: If, while editing, you hear yourself thinking "There ought to be a tool for this" then probably there is one! To find out, use the Zmacs `HELP` command. For example, "There ought to be tools for indenting code."

Press the `HELP` key. The rather cryptic prompt reads:

```
Type one of A,C,D,L,U,V,W,SPACE,HELP,ABORT:
```

Type `A` for "Apropos". This is the way to find out which commands are "apropos" a given topic. We want to find commands related to indenting. When you are prompted for a substring, enter "indent". Then read through the list of commands related to indenting and choose the one that best suits your needs.

For getting information about code you are developing, see the section "Finding Out About Existing Code".

Creating a File Attribute List

Each buffer and generic pathname has attributes, such as Package and Base, which can also be displayed in the text of the buffer or file as an attribute list. An attribute list must be the first nonblank line of a file, and it must set off the listing of attributes on each side with the characters "-*-". If this line appears in a file, the attributes it specifies are bound to the values in the attribute list when you read or load the file.

Suppose you want the new program to be part of a package named **graphics** that contains graphics programs. In this case, you want to set the Package attribute to **graphics** in three places: the generic pathname's property list; the buffer data structure; and the buffer text. You can make the change in two ways:

- If the package already exists in your Lisp environment, use Set Package (m-X) to set the package for the buffer. The command asks you whether or not to set the package for the file and attribute list as well. You cannot use this command to create a new package.

- Use Update Attribute List (m-X) to transfer the current buffer attributes to the file and to create a text attribute list. Edit the attribute list, changing the package. Use Reparse Attribute List (m-X) to transfer the attributes in the attribute list to the file and the buffer data structure. If the package you specify by editing the attribute list does not exist in your Lisp environment, Reparse Attribute List asks you whether or not to create it under **global**.

The mode line of Lisp source files (the line marked by -*-) contains the Base and Syntax attributes. The base can be either 8 or 10 (default). The syntax of a program can be either Zetalisp or Common-Lisp. The defaults for these attributes are as follows:

- If there is a Base attribute, but no Syntax attribute, the syntax defaults to Common-Lisp.

- If there is a Syntax attribute of Common-Lisp, and no Base attribute, the base is assumed to be 10.

- If there is neither a Base nor a Syntax attribute, Base is assumed to be the default base (10) and the syntax is assumed to be Common-Lisp. Furthermore, a warning is issued to the effect that there is neither a Syntax nor a Base attribute. You should edit your program accordingly. With most programs, the Zmacs command Update Attribute List (m-X) will add the appropriate attributes to the mode line, following the above defaults.

When you specify a package by editing the attribute list, you can explicitly name the package's superpackage and, if you want, give an initial estimate of the number of symbols in the package. (If the number of symbols exceeds this estimate, the name space expands automatically.) Instead of typing the name of the package, type a representation of a list of the form (*package superpackage symbol-count*).

To indicate that the **graphics** package is inferior to **global** and might contain 1000 symbols, type into the attribute list:

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

For more on file and buffer attributes, see the section "Buffer and File Attributes in Zmacs".

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.

Command Summary

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, Lisp Syntax, Tab Width, or Vsp.

Update Attribute List (m-X)
Assigns attributes of the current buffer to the associated file and the text attribute list.

Reparse Attribute List (m-X)
Transfers attributes from the text attribute list to the buffer data structure and the associated file.

Zmacs Major and Minor Modes

Each Zmacs buffer has a major mode that determines how Zmacs parses the buffer and how some commands operate. Lisp Mode is best suited to writing and editing

Lisp code. In this major mode, Zmacs parses buffers so that commands to find, compile, and evaluate Lisp code can operate on definitions and other Lisp expressions. Other Zmacs commands, including LINE, TAB, and comment handlers, treat text according to Lisp syntax rules. See the section "Keeping Track of Lisp Syntax in Zmacs".

If you name a file with one of the types associated with the canonical type **:lisp**, its buffer automatically enters Lisp Mode. Here are some examples of names of files of canonical type **:lisp**:

Host system	*File name*
Symbolics	`acme-blue:>symbolics>examples>arrow.lisp`
TOPS-20	`acme-20:<symbolics.examples>arrow.lisp`
UNIX	`acme-vax:/symbolics/examples/arrow.l`

You can also specify minor modes, including Electric Shift Lock Mode and Atom Word Mode, that affect alphabetic case and cursor movement. Whether or not you use these modes is a matter of personal preference. If you want Lisp Mode to include these minor modes by default, you can set a special variable in an init file. If you want to exit one of these modes, simply repeat the extended command. The command acts as a toggle switch for the mode.

Example

The following code in an init file makes Lisp Mode include Electric Shift Lock Mode if the buffer's Lowercase attribute is **nil**, as it is by default:

```
(login-forms
  (setf zwei:lisp-mode-hook
        'zwei:electric-shift-lock-if-appropriate))
```

Command Summary

Lisp Mode (M-X)
: Treats text as Lisp code in parsing buffers and executing some Zmacs commands.

Electric Shift Lock Mode (M-X)
: Places all text except comments and strings in uppercase.

Atom Word Mode (M-X)
: Makes Zmacs word-manipulation commands (such as M-F) operate on Lisp symbol names.

Auto Fill Mode (M-X)
: Automatically breaks lines that extend beyond a preset fill column.

Set Fill Column: C-X F
: Sets the fill column to be the column that represents the current cursor position. With a numeric argument less than 200,

sets the fill column to that many characters. With a larger numeric argument, sets the fill column to that many pixels.

Keeping Track of Lisp Syntax in Zmacs

Zmacs allows you to move easily through Lisp code and format it in a readable style. Commands for aligning code and features for checking for unbalanced parentheses can help you detect simple syntax errors before compiling.

Zmacs facilities for moving through Lisp code are typically single-keystroke commands with c-m- modifiers. For example, Forward Sexp (c-m-F) moves forward to the end of a Lisp expression; End Of Definition (c-m-E) moves forward to the end of a top-level definition. Most of these commands take arguments specifying the number of Lisp expressions to be manipulated. In Atom Word Mode word-manipulating commands operate on Lisp symbol names; when executed before a name with hyphens, for example, Forward Word (m-F) places the cursor at the end of the name rather than before the first hyphen. See the section "Zmacs Major and Minor Modes".

For a list of common Zmacs commands for operating on Lisp expressions: See the section "Editing Lisp Programs in Zmacs".

Commenting Lisp Code

You can document code in two ways:

- You can supply documentation strings for functions, variables, and constants. For information on how to retrieve those documentation strings with Zmacs commands and Lisp functions: See the section "Finding Out About Existing Code".

- You can insert comments in the source code. The Lisp reader ignores source-code comments. Although you cannot retrieve them in the same ways as documentation strings, they are essential to maintaining programs and useful in testing and debugging. See the section "Compiling and Evaluating Lisp". See the section "Debugging Lisp Programs".

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 start a comment with #|| and end it with ||#. We recommend using #||...||# instead of #|...|# to comment out Lisp code because it is interpreted as a comment by both the Lisp reader and the editor.

#| begins a comment for the Lisp reader. The reader ignores everything until the next |#, which closes the comment. #| and |# can be on different lines, and #|...|# pairs can be nested.

Use of #|...|# always works for the Lisp reader. The editor, however, currently does not understand the reader's interpretation of #|...|#. Instead, the editor retains its knowledge of Lisp expressions. Symbols can be named with vertical bars, so the editor (not the reader) behaves as if #|...|# is the name of a symbol surrounded by pound signs, instead of a comment.

Now consider #||...||#. The reader views this as a comment: the comment prologue is #|, the comment body is |...|. and the comment epilogue is |#. The editor, however, interprets this as a pound sign (#), a symbol with a zero length print name (||), lisp code (...), another symbol with a zero length print name (||), and a stray pound sign (#). Therefore, inside a #||...||#, the editor commands that operate on Lisp code, such as balancing parentheses and indenting code, work correctly.

Example

The following example shows how comments can be used effectively.

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
```

Command Summary

Indent For Comment: c-; or m-;
: Inserts or aligns a comment on the current line, beginning in the preset comment column.

Comment Out Region: c-X c-;
: Comments out the region or definition.

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) or m-Q
: 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.

Aligning Lisp Code in Zmacs

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.

Command Summary

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.

Balancing Parentheses in Zmacs

When the cursor is to the right of a close parenthesis, Zmacs flashes the corresponding open parenthesis. The flashing open parentheses, along with proper indentation, can indicate whether or not parentheses are balanced. Improperly aligned code (after you use a c-m-Q command, for instance) is often a sign of unbalanced parentheses.

To check for unbalanced parentheses in an entire buffer, use Find Unbalanced Parentheses (m-X). Zmacs can check source files for unbalanced parentheses when you save the files. If a file contains unbalanced parentheses, Zmacs can notify you and ask whether or not to save the file anyway. To put this feature into effect, place the following code in an init file:

```
(login-forms
  (setf zwei:*check-unbalanced-parentheses-when-saving* t))
```

Command Summary

Find Unbalanced Parentheses (m-X)
Searches the buffer for unbalanced parentheses. Ignores parentheses in comments and strings.

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.

Command Summary

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.

Searching and Replacing

Some facilities discussed elsewhere, particularly the series of List and Edit commands, are useful for displaying and moving to code you wish to edit. See the section "Finding Out About Existing Code".

The commands we discuss here find and replace strings. *Tag tables* offer a convenient means of making global changes to programs stored in more than one file. Use Select All Buffers As Tag Table (m-X) to create a tag table for all buffers read in. Use Select System As Tag Table (m-X) to create a tag table for all files in a system. For information on systems: See the section "System Construction Tool".

Command Summary

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 incremental Search c-R
: Prompts for a string and moves backward to its last occurrence in the buffer. Press c-R to repeat the search with the same string. Press c-S to search forward with the same string. After you invoke the command, if c-R is the first character you type (instead of a string), uses the string specified in the previous search.

Tags Search (m-X) Searches for a string in all files listed in a tag table.

Replace String (m-X) or c-?
: In the buffer, replaces all occurrences (following point) of one string by another.

Query Replace (m-X) or m-?
: In the buffer, replaces occurrences (following point) of one string by another, querying before each replacement. Press HELP for possible responses.

Tags Query Replace (m-X)
: In files listed in a tag table, replaces occurrences of one string by another, querying before each replacement.

Select All Buffers As Tag Table (m-X)
: Creates a tag table for all buffers in Zmacs.

Select System As Tag Table (m-X)
: Creates a tag table for files in a system defined by **defsystem**.

Moving Text

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.

Command Summary

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 or c-@
: 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.

Killing and Yanking

When you need to repeat text, you usually want to copy it rather than type it again. The most common facilities for copying text are the commands for killing and yanking. Commands that kill more than one character of text push the text onto the kill ring. c-Y yanks the last kill into the buffer. After a c-Y command, m-Y deletes the text just inserted, yanks the previous kill, and rotates the kill ring.

Example

Suppose we have defined a function in one program, and we want to copy the function to another buffer. We can copy the function in three ways:

- Use c-m-K or c-m-RUBOUT to kill the definition. Use c-Y to restore it. Go to the new buffer. Use c-Y to insert a copy of the definition.

- Use c-m-H to mark the definition. Use m-W to push it onto the kill ring. Go to the new buffer. Use c-Y to insert a copy of the definition.

- Click middle on the first or last parenthesis of the definition to mark the definition. Click sh-Middle to push it onto the kill ring. Move to the new buffer. Click sh-Middle to insert a copy of the definition.

Command Summary

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* (sh-Middle)]
When *region* is defined, pushes the text of *region* onto the kill ring without killing the text (like m-W). Repeated execution has the following effects:

- First repetition: kills the text of *region*, pushing the text onto the kill ring (like c-W)
- Second repetition: pops the text of *region* from the kill ring, inserting the text into the buffer at point (like c-Y)
- Third and subsequent repetitions: deletes the text just inserted, yanks previously killed text from the kill ring, and rotates the kill ring (like m-Y)

If no region is defined, pops the last killed text from the kill ring, inserting the text into the buffer at point (like c-Y). Repeated execution deletes the text just inserted, yanks previously killed text from the kill ring, and rotates the kill ring (like m-Y).

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.

Command Summary

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.

Inserting Buffers and Files Into a Buffer

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.

Command Summary

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) or c-X I
Inserts contents of the specified file into the current buffer at point.

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).

Command Summary

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 sh-Left). 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.

Zmacs Buffers in Multiple Windows

Sometimes when editing you move often between two buffers. You might want to see the two buffers at the same time rather than switch between them. A common use of multiple-window display is to edit source code while viewing compiler warnings. See the section "Using the Compiler Warnings Database".

Example

After developing a 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 displaying the bug-report file. We edit the program code, using c-m-V to scroll the bug-report window as we correct each bug.

Command Summary

Split Screen (m-X) Pops up a menu of buffers and splits the screen to display the buffers you select.

Two Windows: c-X 2
: Creates a second window, with the current buffer on top and the previous buffer on the bottom. Puts the cursor in the bottom window.

View Two Windows: c-X 3
: Creates a second window, with the current buffer on top and the previous buffer on the bottom. Puts the cursor in the top window.

Modified Two Windows: c-X 4
: Creates a second window and visits a buffer, file, or tag there. Displays the current buffer in the top window.

Other Window: c-X O
: Moves to the other of two windows.

Scroll Other Window: c-m-V
Scrolls the other of two windows.

One Window: c-X 1 Returns to one-window display, selecting the buffer the cursor is in.

Finding Out About Existing Code

Programmers can read this section to learn about Zmacs and Genera tools for retrieving information about Lisp objects (including symbols, variables, functions, and pathnames), and for displaying and editing source code.

When you write or edit programs, you often need to find characteristics of existing code. If you write programs incrementally, you need to find existing definitions, argument lists, and values. To maintain modularity, you must know how new code should interact with previously written modules. If you want to incorporate parts of the Symbolics system in your programs, you often have to refer to system source code.

Many of the examples in the sections that follow are based on the Life program: See the section "The Life Program in Common Lisp".

Finding Out About Objects

describe displays information about a Lisp object in a form that depends on the object's type. For example, for a special variable, **describe** displays the value, package, and properties, including documentation, pathname of the source file, and Zmacs buffer sectioning node.

The Inspector is an interactive, window-oriented version of **describe**. See the section "Using the Inspector".

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

Example

```
(describe '*number-cells-on-axis*)

The value of *NUMBER-CELLS-ON-AXIS* is 30
*NUMBER-CELLS-ON-AXIS* is in the USER (really COMMON-LISP-USER) package.
*NUMBER-CELLS-ON-AXIS* has property :DOCUMENTATION:
(DEFVAR "Number of cells on each axis of game board")
    (DEFVAR "Number of cells on each axis of game board") is a list
```

```
*NUMBER-CELLS-ON-AXIS* has property SPECIAL: T
*NUMBER-CELLS-ON-AXIS* has property :SOURCE-FILE-NAME: ((DEFVAR #))
    ((DEFVAR #)) is a list

*NUMBER-CELLS-ON-AXIS* has property ZWEI:ZMACS-BUFFERS: ((DEFVAR #))
    ((DEFVAR #)) is a list

*NUMBER-CELLS-ON-AXIS*
```

Reference

describe *anything* &optional *si:*describe-no-complaints**
Provides all the interesting information about any object (except array contents).

zl:listarray *array* &optional *limit*
Creates and returns a list whose elements are those of *array*.

Finding Out About Symbols

Several Zmacs commands and Lisp functions find the name of a symbol or retrieve information about it. Unless you specify a package, most of these commands search the **global** package and its inferiors. For more on the meanings and default values of arguments to these functions: See the section "Getting Help".

Example

In defining the function **calculate-next-board**, we need to use the constants ***game-board*** and ***next-game-board***, but we remember only that their names contains "board". We use either `m-ESCAPE` (to evaluate a form in the editor minibuffer) or `SELECT L` (to select a Lisp Listener) and then use the **apropos** function:

```
(apropos "board" 'cl-user)

Searching the package USER (really COMMON-LISP-USER),
including inherited symbols, for symbols containing
the substring "board".
```

```
CL-USER::*NEXT-GAME-BOARD* - Bound
CL-USER::CURRENT-BOARD
CL-USER::BOARD
CL-USER::INITIALIZE-BOARD-WITH-CELLS - Function (BOARD)
CL-USER::DISPLAY-GAME-BOARD - Function (&optional (BOARD *GAME-BOARD*))
CL-USER::NEXT-BOARD
CL-USER::CALCULATE-NEXT-BOARD - Function ()
CL-USER::*GAME-BOARD* - Bound
CL-USER::MAKE-GAME-BOARD - Function ()
Done; the value of *APROPOS-LIST* is a list of the symbols found.
```

We can find out which function calls **initialize-board-with-cells** by using **who-calls**:

```
(who-calls 'initialize-board-with-cells)

PLAY-LIFE-GAME calls INITIALIZE-BOARD-WITH-CELLS as a function.
(PLAY-LIFE-GAME)
```

You can also find the callers of a function with List Callers (m-X). See the section "Finding Out About Lisp Functions".

Command and Function Reference

apropos *string* &optional *package (do-inherited-symbols* **t***) do-packages-used-by*
: Searches for all symbols whose print-names contain *string* as a substring.

who-calls *symbol* &optional *how*
: Tries to find all the functions in the Lisp world that call *symbol*.

Where Is Symbol (m-X)
: Displays the names of packages that contain the specified symbol.

where-is *pname*
: Finds all symbols named *pname* and prints a description of each symbol.

what-files-call *symbol-or-symbols* &optional *how*
: Returns a list of the pathnames of all the files that contain functions that **who-calls** would have printed out.

symbol-plist *symbol*
: Returns the list that represents 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.

Finding Out About Variables

Describe Variable At Point (c-sh-V) is a useful command to display information about a variable. It tells you whether or not the variable is bound, whether it has been declared special, and the file, if any, that contains the declaration. You can find the value of a variable by evaluating it in a Lisp Listener. If you have added a documentation string to the variable declaration, you can retrieve the string with c-sh-V or with c-sh-D, m-sh-D, or **documentation**. See the section "Finding Out About Lisp Functions".

Example

We might want to find out whether ***number-cells-on-axis*** is bound. In the editor buffer, we position the cursor on that variable, where it occurs in a function definition. c-sh-V displays the following information:

```
*number-cells-on-axis* has a value and is declared special
```

Command Summary

Describe Variable At Point: c-sh-V
: Indicates whether or not the variable is declared special, is bound, or is documented by **defvar** or **zl:defconst**.

Finding Out About Lisp Functions

Many Zmacs and Genera 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".

Tools for getting information about flavors, generic functions, and methods are described elsewhere: See the section "Flavors Tools".

Finding Out About Lisp Definitions

Edit Definition (m-.) is a powerful command for finding and editing definitions of functions and other objects. It is particularly valuable for finding source code, including system code, that is stored in a file other than that associated with the current buffer. It finds multiple definitions when, for example, a symbol is defined as a function, a variable, and a flavor. It maintains a list of these definitions in a support buffer, where you can use m-. to return to the definitions even when you are finished editing.

For a description of how to use Edit Definition (m-.) to edit definitions of flavor methods: See the section "Finding Out About Methods".

Command Summary

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.

Finding Out About Function 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 **initialize-board-with-cells** from **play-life-game**, but we remember only that **initialize-board-with-cells** contains the string "initialize-board". We use Function Apropos (m-X) to display the names of functions that contain "intialize-board":

```
m-X Function Apropos initialize-board

Functions matching arrow:
INITIALIZE-BOARD-WITH-CELLS
```

We could click Left on the name **initialize-board-with-cells** to edit its definition.

Command Summary

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.

Finding Out About Documentation Strings

Function definitions can include documentation strings. When you need to know the purpose of the function, you can retrieve the documentation with c-sh-D, m-sh-D, or **documentation**.

Example

We defined **initialize-board-with-cells** with a documentation string:

```
(defun initialize-board-with-cells (board)
  "Initialize inner part of the array with cells
Cells are randomly chosen to be alive or dead."
  (do ((y 1 (+ 1 y)))
      ((= y *number-cells-on-axis*))
    (do ((x 1 (+ 1 x)))
        ((= x *number-cells-on-axis*))
      (setf (aref board x y)
            (if (evenp (random 2)) 1 0)))))
```

Later, when defining **play-life-game**, we know we want to call **initialize-board-with-cells**. If at that time, we want to read its documentation string, we can position the cursor at the name **initialize-board-with-cells** inside the definition of **play-life-game** and use `c-sh-D`:

The summary documentation is displayed in the echo area at the bottom of the screen:

```
INITIALIZE-BOARD-WITH-CELLS:  (BOARD)
INITIALIZE-BOARD-WITH-CELLS:  Initialize inner part of the array with cells
```

The complete documentation string is displayed in the typeout window at the top of the screen:

```
Initialize inner part of the array with cells
Cells are randomly chosen to be alive or dead.
```

Command Summary

Long Documentation: `c-sh-D`
: Displays the function's documentation string.

documentation *name* &optional *(type* **'defun***)*
: Finds the documentation string of the symbol, *name*, which is stored in various different places depending on the symbol type.

Show Documentation (`m-X`) or `m-sh-D`
: Displays the function's documentation, if it is documented as a topic available in Document Examiner.

Finding Out About Argument Lists

Quick Arglist (`c-sh-A`) and **arglist** retrieve the argument list for an ordinary function, a generic function, or a **send** form with a constant message name. The output displayed depends on the nature of the function, whether or not it has been compiled, and what options the function includes. For details:

See the function **arglist**.
See the section "Getting Help".

Example

We are editing the definition of **play-life-game** to add a call to **initialize-board-with-cells**. To see the argument list for **initialize-board-with-cells**, we position the cursor at the name **initialize-board-with-cells** in the definition of **play-life-game** and use c-sh-A:

```
INITIALIZE-BOARD-WITH-CELLS:  (BOARD)
```

In the Lisp Listener, when we are ready to call **play-life-game**, suppose we forget the arguments of the function. We can type this much and use c-sh-A:

```
(play-life-game
```

The arguments are displayed as follows:

```
PLAY-LIFE-GAME: (&optional (GENERATIONS 3))
```

Command Summary

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* &optional *real-flag arglist-finder*
: Returns a representation of the lambda-list of *function*.

Finding Out About Callers

When you change a function definition, you sometimes need to make complementary changes in the function's callers. For example, if you change the order of arguments to a function, you need to change the order of arguments in its callers.

Four Zmacs commands find the callers of a function. By default, these commands search the current package for callers. (For some programs, it is necessary to search only in the package of the program itself.) 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.

Command Summary

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.

Finding Out About Methods

For more information about tools that describe flavors, generic functions, and methods: See the section "Flavors Tools".

You can use the m-. Zmacs command 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:

(flavor:method *generic-function flavor options..***)**

For information on other flavor-related function specs: See the section "Function Specs for Flavor Functions".

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 generic function. You can click on one of the method names displayed to edit its definition.

Command Summary

List Methods (m-X)
Prompts you for a generic function name (or message name). Lists the methods of all flavors that handle the generic function, in a mouse-sensitive display. To edit one of the methods, click on its function spec in the display. Alternatively, you can use c-. to edit a method. c-. cycles through the methods, each time choosing the next method for you to edit.

Edit Methods (m-X)
Prompts you for a generic function name (or message name). One of the definitions is found and pulled into an editor buffer. c-. cycles through the methods, each time choosing the next method for you to edit.

If more than one definition is available, the list of definitions and their source files is stored in a buffer *METHODS-*n*.

List Combined Methods (m-X)
Prompts for a generic function name, then for a flavor name. It then lists the methods for the specified generic function when applied to the specified flavor. This is a mouse-sensitive display. To edit one of the methods, click on its function spec

in the display. Alternatively, you can use c-. to edit a method. c-. cycles through the methods, each time choosing the next method for you to edit.

Error messages appear if the flavor does not handle the generic function, or if the flavor requested is not a composed, instantiated flavor.

List Combined Methods (m-X) can be useful for predicting what a flavor will do in response to a generic function. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them.

Edit Combined Methods (m-X)
: Asks you for a generic function name and a flavor name. This command finds all the methods that are called if that generic function is called on an instance of the given flavor. You can point to the generic function and flavor with the mouse; completion is available for the flavor name. As in Edit Methods (m-X), the command skips the display and proceeds directly to the editing phase.

Finding Out About Pathnames

Zmacs provides several ways of finding the name of a file. If you just need the name of a file and have some idea what directory it is in, you can use c-X c-D with an argument of c-U, or Show 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.

Command Summary

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.

Show Directory (m-X)
: Lists a directory.

Dired: c-X D
: Edits the directory of the file in the current buffer. 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.

Compiling and Evaluating Lisp

When should you compile code, and when evaluate it?

The main job of the compiler is to convert interpreted functions into compiled functions. An interpreted function is a list whose first element is **lambda**, **zl:named-lambda**, **zl:subst**, or **zl:named-subst**. These functions are executed by the Lisp evaluator. The most common interpreted functions you define are **zl:named-lambda**s. When you load a source file that contains **defun** forms or when you otherwise evaluate these forms, you create **zl:named-lambda** functions and define the function specs named in the forms to be those functions.

Compiled functions are Lisp objects that contain programs in the instruction set (the machine language). They are executed directly by the microcode. Compiling an interpreted function (by calling the compiler on a function spec) converts it into a compiled function and changes the definition of the function spec to be that compiled function.

You seldom compile functions directly. Instead, you compile either regions of Zmacs buffers or source files.

- Compiling a region of a Zmacs buffer (or the whole buffer) causes the compiler to process the forms in the region, one by one. This processing has side effects on the Lisp environment. For a summary of the compiler's actions: See the section "Compiling Code in a Zmacs Buffer".

- Compiling a source file translates it into a binary file. With some exceptions, this processing does not have side effects on the Lisp environment at compile time. When you load a compiled file that defines functions, you create compiled rather than interpreted functions and define function specs to be those compiled functions. In other respects, loading a compiled file has essentially the same effects as loading a source file (evaluating the forms in the file). For a discussion of compiling files: See the section "Compiling and Loading a File".

Most programmers compile all their program code. The compiler checks extensively for errors and issues warnings that help detect bugs like typographical errors, unbound symbols, and faulty Lisp syntax. Compiled code runs faster and takes up less storage than interpreted code. You can compile code in portions and decide at compile time whether or not to save the compiler output in a binary file.

The most common use for interpreted functions is stepping through their execution. You cannot step through the execution of a compiled function. If a function is compiled, you can read its definition into a Zmacs buffer, evaluate the definition, and then step through a function call.

In addition to evaluating definitions to create interpreted functions, you might need to evaluate forms to test a program or find information about a Lisp object. (Unless you are using the Stepper, functions that you call during these evaluations are usually compiled.) You can evaluate a form in a Lisp Listener, a breakpoint loop, or the minibuffer.

For more information on functions: See the section "Functions".

Compiling Lisp Code

You can use Zmacs commands to compile code in a file or Zmacs buffer. You can compile as soon as you have written enough to test. This practice lets you 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.

Often it is useful to recompile a function soon after making a change. Because recompiling a series of functions or an entire program can be time-consuming, it is easier and faster to make changes and then use a single command to recompile only the changed functions. Using Compile Changed Definitions Of Buffer (m-sh-C) or Compile Changed Definitions (m-X) is easier in this case than recompiling each function separately or recompiling the entire buffer.

The order in which you compile definitions can be important. For example, suppose you have a function that binds a variable you want to be treated as special. If you compile the function definition before compiling the variable declaration, the compiler treats the variable as local and generates incorrect output. For this reason you should usually put **defvar** and **zl:defconst** forms at the beginning of a file or into a separate file to be compiled and loaded before function definitions.

When editing a program, it is a good idea to load the entire program before you start work on it. When you compile new definitions or recompile edited ones, the compiler will have access to variable declarations, macros, functions, and other information. You will also be able to use Zmacs commands and Lisp functions for finding information about other parts of the program. See the section "Finding Out About Existing Code".

Sometimes when you compile a file, write large sections of code at once, or make many changes to a large program, compiling the code produces many warning messages. For a description of how Edit Compiler Warnings (m-X) lets you use the compiler warnings as a reference source for debugging: See the section "Debugging Lisp Programs".

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

Compiling Code in a Zmacs Buffer

Compiling a top-level form in a Zmacs buffer — using a command such as 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 defi-

	nition, converts the lambda-expression into a compiled function, and defines the function spec named in the definition to be that compiled function.
Macro definition	If the form is a list whose first element is **macro**, the compiler constructs a lambda-expression as the macro's expander function, converts the lambda-expression into a compiled function, and defines the function spec named in the definition to be the macro. A **defmacro** form expands into this kind of form.
Special case	Some forms, like **eval-when**, **declare**, and **progn 'compile** forms, have special meaning for the compiler. It handles each of these in a different way. For details: See the section "How the Stream Compiler Handles Top-level Forms".
Atom, **zl:comment** form	The form is ignored.
Other	The form is evaluated.

Command Summary

Compile Region (m-X) or c-sh-C
: Compiles the region. If no region is marked, compiles the current definition.

Compile Changed Definitions Of Buffer (m-X) or 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.

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** or **:ibin**. The type **:ibin** is for files compiled for Ivory-based machines, and **:bin** is for files compiled for 3600-family machines.

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 com-

pile a file and then load the resulting binary file by using **compiler:compile-file-load**.

Command and Function Summary

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** or **:ibin**.

compile-file *input-file* &key *:output-file :package :load (:set-default-pathname* ***compile-file-set-default-pathname****)*
Compiles a file and places the compiled code in a file of canonical type **:bin** or **:ibin**.

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 do not 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 *filename* &key *(:verbose* ***load-verbose****) :print (:if-does-not-exist* **:error***) :package :default-package (:set-default-pathname* ***load-set-default-pathname****)*
Loads the file specified by *filename* into the Lisp environment.

Evaluating Lisp Code

Evaluating Lisp Code Within Zmacs

The most common reason for evaluating definitions in a Zmacs buffer is to step through the execution of the functions they define. Sometimes in debugging you want to proceed step by step through a function call, using **zl:step** or the **:step** option for **trace**. See the section "Tracing and Stepping".

You can step through the execution of a function only if it is an interpreted function. 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 to select a Lisp Listener and evaluate forms there. Use SELECT E to return to the editor.

Command Summary

Evaluate Region (m-X) or c-sh-E
: Evaluates the region. If no region is marked, evaluates the current definition.

Evaluate Changed Definitions Of Buffer (m-X) or 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.

Short-cuts for Evaluating Lisp Code in the Lisp Listener

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.

`c-m-Y` Yanks the last form typed to the Lisp Listener. It waits after the final delimiter for you to press `END`, allowing you to edit the form before evaluating it. With an argument *n*, yanks the *n*th form in the input ring. In Zmacs, this command performs a different action: it repeats the last minibuffer command typed.

`m-Y` After a `c-m-Y` command, deletes the form just inserted, yanks the previous form from the input ring, and rotates the input ring. Repeated execution yanks previous forms and rotates the input ring. In Zmacs, this command rotates either the minibuffer command history or the text kill history (depending on which yanking command it follows), and yanks elements from that history. See the section "Retrieving History Elements".

Debugging Lisp Programs

The Genera software environment offers a powerful interactive Debugger and a variety of other tools for debugging Lisp programs. The kind of debugging tool you use depends on the kind of program. Bugs might be more obvious in a graphics programs than in a minor modification of some internal system function. Problems with a graphics programs are sometimes evident from the program's output. On the other hand, programs with a complex window system application might have bugs that are difficult to identify.

Debugging tools are more appropriate for some kinds of bugs than for others. You commonly encounter three sorts of problems with a program:

- The program does not compile correctly. You can use the compiler warnings database to edit code before recompiling.

- The program compiles, but running it signals an error. Usually errors invoke the Debugger, where you can examine stack frames, return values, disassemble code, call the editor, and perform other tasks.

- The program runs but does not behave as it should. You can use many techniques for finding the problem, including commenting out sections of code, trac-

ing, stepping, setting breakpoints, disassembling, and inspecting. Often the most effective method is simply studying the source code.

For complete information on the Debugger: See the section "Debugger". Also: See the section "Miscellaneous Debugging Techniques".

Using the Compiler Warnings Database

The compiler sometimes produces many warning messages. The compiler maintains a database of these messages, organized by file. Each time you compile or recompile code, the compiler adds or removes warnings from the database, so that the database reflects the state of your program as of the last time you compiled it.

If you want to save warnings in a file, you can use Compiler Warnings (m-X) to put them in a buffer and then write them to a file.

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.

Command Summary

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.

Overview of Using the Debugger

This section gives a brief introduction to using the Debugger. For complete documentation on this subject: See the section "Using the Debugger".

Some errors during execution automatically invoke the Genera Debugger. You can also enter the Debugger explicitly by pressing m-SUSPEND or c-m-SUSPEND. You can also enter the Debugger from within a program by inserting a call to **zl:break** or **zl:dbg** with no arguments into the code and recompiling. You can force a process into the Debugger by calling **zl:dbg** with an argument of *process*.

See the section "Using Breakpoints".

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 entering the :Window Debugger command or by pressing c-m-W. See the section "Using the Display Debugger".

An error invokes the Debugger with the name of the function in which the error occurred, the value of the function's arguments, and an error message such as:

```
>Trap: The first argument given to SYS:--INTERNAL, NIL, was not a number.
```

The Debugger also displays a listing of *proceed types*, *special commands*, and *restart handlers*, along with their key bindings: See the section "Special Keys". We can use one of these options, or we can use other Debugger commands to examine or manipulate the stack.

Summary of Useful Debugger Commands

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

:Show Argument (c-A)
: Shows arguments for the current stack frame.

:Edit Function (c-E)
: Calls the editor to edit the function from the current frame.

:Show Frame (c-L) Clears the screen and redisplays the original error message.

:Bottom Of Stack (m->)
: Moves to the bottom of the stack and displays the least-recent frame.

:Next Frame (c-N) Moves down the stack by one frame.

:Previous Frame (c-P)
: Moves up the stack by one frame.

:Top Of Stack (m-<) Moves to the bottom of the stack and displays the most recent frame.

:Return (c-R) Returns a value from the current frame.

:Show Backtrace (m-B)
Shows a backtrace of function names with arguments.

:Show Local (c-m-L)
Shows local variables and disassembled code for the current frame.

:Reinvoke (c-m-R) Reinvokes the current frame.

:Window Debugger (c-m-W)
Invokes the Window Debugger.

:Show Compiled Code (c-X D)
Displays the disassembled code for a function.

:Show Source Code (c-X c-D)
Displays the source code for a function.

:Analyze Frame (c-m-Z)
Analyzes the erroneous frame and locates the source code of the current error.

:Describe Last (c-m-D)
Executes the Lisp **describe** function on the most recently displayed value and leaves * set to that value.

:Show Special Displays the special-variable binding of a symbol in the context of the current frame.

Using the Display Debugger

The Display Debugger is a version of the standard Debugger that uses its own multi-paned window. You enter the Display Debugger from the standard Debugger by pressing c-m-W, or by using the :Window Debugger command.

Figure ! shows the Display Debugger, entered while in the Concordia editor.

Overview

The Display Debugger divides the screen into eight panes, showing various aspects of the program environment at the time of the error. It displays the name of the activity you are debugging in the top left pane.

When the Display Debugger is entered, the various panes contains the state of the erring frame. Most common operations are available directly from the mouse.

In addition to using the commands listed in the Command Menu in the Display Debugger, you can also use all of the usual Debugger commands by typing them on the keyboard.

Display Debugger Panes

Here is a description of each pane in the Display Debugger:

The Proceed Options pane -- the top righthand pane

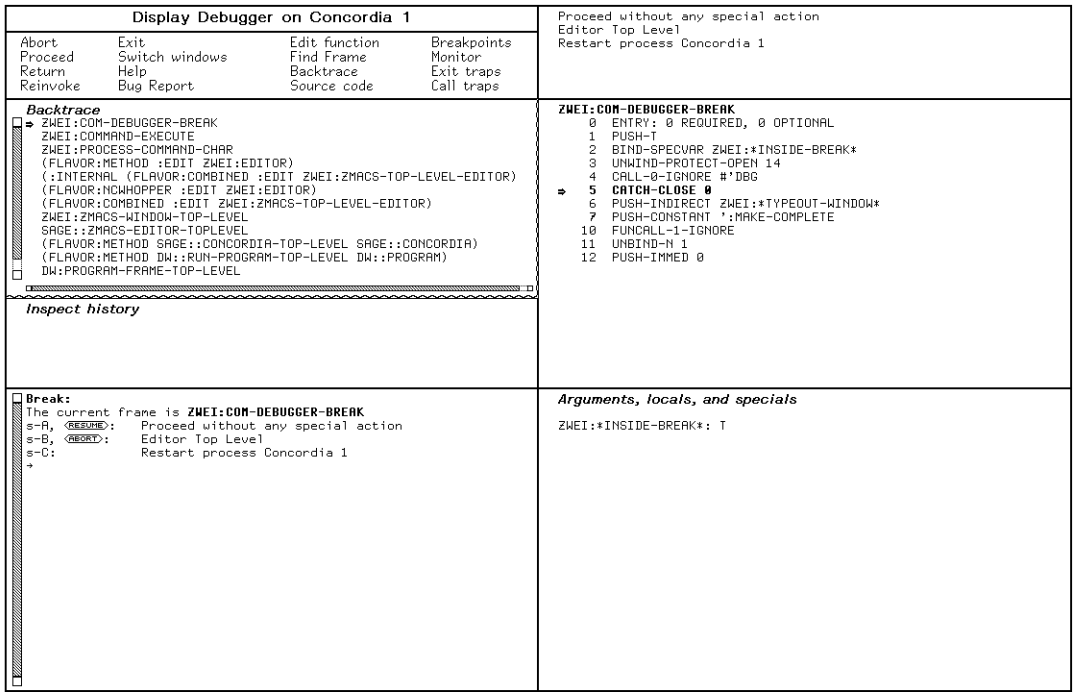

Figure 85. The Display Debugger

The proceed options displayed in this pane are the same as seen in the standard Debugger. Clicking Left on a proceed option takes that option. See the section "Using Debugger Proceed and Restart Options".

The Code pane -- the middle righthand pane

This pane displays the source code of the erring frame. If the Debugger cannot find the source code, it displays the disassembled code instead.

If you compiled the code with source locators, the form that caused the Debugger to be entered is highlighted in boldface. Furthermore, the forms in the source code are sensitive as pieces of Lisp code. This means that you can click Left on a form in the source code to evaluate it in the context of the erring frame, or you can click c-m-Left on a form to set a breakpoint at that place in the code.

If the Display Debugger is showing disassembled code, the current PC is highlighted in boldface with an arrow. You can click Left on the names of local and special variables to see their values, or you can click c-m-Left on a PC in the disassembly to set a breakpoint at that PC.

Arguments, locals and specials pane -- the bottom righthand pane

This pane shows you all of the arguments and locals for the current frame, as well as any special variables bound in the frame. You can describe an argument, local, or special by clicking Middle on it. You can inspect it (with the Inspector) by clicking sh-Middle on it. You can modify the arguments, locals, or specials by pressing clicking c-m-Right on it.

Title pane -- the top lefthand pane

The Title pane shows you the name of the activity that you are debugging.

Command menu pane -- the second lefthand pane

This pane contains mouse-sensitive commands to do various things such as change your activity or placement in the Display Debugger.

If you position your cursor on one of the command names, the mouse documentation line shows what happens if you click Left, Middle, or Right on a particular command.

Here is a brief description of each of the commands in this pane:

Abort	Leaves the Display Debugger and aborts from the error.
Proceed	Proceeds from the error using the Resume proceed handler.
Return	Returns from the current frame. If you click Middle on this command, the Display Debugger asks you which frame you wish to return from.
Reinvoke	Reinvokes the current frame. If you click Middle on this command, the Display Debugger asks you which frame you wish to reinvoke.
Exit	Exits from the Display Debugger back to the standard Debugger. This neither aborts nor proceeds from the error.
Switch Windows	Switches to the original window where the error occurred. Press `FUNCTION S` to return to the Display Debugger.
Help	Shows a brief help display.
Bug Report	Allows you to create, edit, and mail a bug report.
Edit function	Edits the function for the current frame.
Find Frame	Searches down from the current frame for one whose function name contains a specified substring. Clicking Middle on the command causes the last search to be executed again from the new starting place.
Backtrace	Click Left on this command to display an "ordinary" backtrace, one that hides invisible frames. Click Middle on this command to display a backtrace which does not hide invisible frames.
Source Code	Click Left on this command to see the source code for the frames, if it is available. Click Middle on this command to see disassembled code for the frames.
Breakpoints	Click Left for a display of all the currently set breakpoints. Click Middle to clear all the breakpoints. Click Right to get a menu of various other breakpoint-related commands.
Monitor	Click Left for a display of all the currently monitored locations. Click Right for a menu of various other monitor-related commands.

Exit traps	Click Left to set trap-on-exit for the current frame. Click Middle to clear trap-on-exit for the current frame. Click Right for a menu of various other trap-on-exit commands.
Call traps	Click Left to set trap-on-call for the current frame. Click Middle to clear trap-on-call for the current frame. Click Right for a menu of various other trap-on-call commands.

Backtrace pane -- the third lefthand pane

This pane displays the backtrace for the current error. The current frame is indicated by an arrow on the left. Clicking Left on a frame in this pane causes the current frame to be set to that frame. Clicking Middle on a frame shows the arguments with which that frame was called. Clicking Right on a frame pops up a menu for all the operations on that frame, such as set or clear trap-on-exit, disassemble the function for the frame, edit the frame's function, reinvoke this frame, return from this frame, and so forth.

Inspect history pane -- the fourth lefthand pane

This pane keeps track of all of the "complex" Lisp objects that you have examined. You can reexamine objects in this pane by clicking Middle or ⇧-Middle on them.

Interactor pane -- the bottom lefthand pane

This pane is where interaction with the Display Debugger takes place. You can use all the usual Debugger commands and accelerators in this pane. For a list of Debugger commands: See the section "Debugger Command Descriptions".

Commenting Out Code

Sometimes a program runs but behaves in an unexpected way. In looking for the source of the problem, you might want to execute some portions of the program and disable others. An easy way to disable code without destroying it is to make a comment of it. You can comment out code by preceding it with a semicolon or surrounding it with #||...||#: See the section "Commenting Lisp Code".

Tracing and Stepping

Tracing

When a program runs but behaves unexpectedly, you might be calling functions in the wrong sequence or passing incorrect arguments. Tracing function calls can help detect this sort of problem. By default, tracing prints a message, indented according to the level of recursion, on entering and leaving a function. It also prints the arguments passed and the values returned.

You can invoke tracing in three ways:

- Use [Trace] in the System menu

- Use Trace (m-X) in Zmacs
- Use the **trace** special form

[Trace] and Trace (m-X) pop up a menu of options, including stepping and inserting breakpoints. You can use these options with **trace**, too, but the syntax is complex. Table ! summarizes the correspondence between trace menu items and **trace** options. For a description of the options: See the section "Options to **trace**".

Table 11. Trace Menu Items and trace Options

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

Command and Function Summary

Trace (M-X) — Traces or untraces a specified function. Prompts for the name of a function to trace and pops up a menu of trace options.

[Trace] (from the System menu) — Traces or untraces a specified function. Prompts for the name of a function to trace and pops up a menu of trace options.

(**trace** (**:function** *function-spec-1 option-1 option-2* ...) ...) — Enables tracing of one or more functions. If *function-spec* is a symbol, the keyword **:function** is unnecessary. An argument can also be a list whose car is a list of function names and whose cdr is one or more options. In this case, all functions in the list are traced with the same options.

(**trace**) — When called with no arguments, **trace** 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.

(**untrace**) — When called with no arguments, **untrace** disables tracing of all functions being traced.

Stepping

When a program behaves unexpectedly and tracing doesn't reveal the problem, you might step through the evaluation of a function call. You can step through function execution by using **step**, [Step] from a trace menu, or the **:step** option for **trace**.

You can step through the execution of a function only if it is interpreted, not compiled. If you want to step through execution of a compiled function, read the definition into a Zmacs buffer and use a Zmacs command (such as C-sh-E) to evaluate it. See the section "Evaluating Lisp Code Within Zmacs".

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. For a list of commands, press HELP inside the Stepper, or: See the section "Stepping Through an Evaluation".

Summary of Basic Stepper Commands

Command	*Action*
c-N	Evaluate until next thing to print
SPACE	Evaluate until next thing to print at this level (don't step at lower levels)
c-U	Evaluate until next thing to print at next level up (don't step at current and lower levels)
c-B	Enter breakpoint loop
c-E	Enter Zmacs
c-X	Evaluate until finished (exit from stepping)

Command and Function Summary

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

Trace (m-X) [Step] Steps through the execution of a function being traced.

[Trace / Step] (from the System menu)
Steps through the execution of a function being traced.

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

Using Breakpoints

In debugging a program, you might want to interrupt function execution to enter a Lisp breakpoint loop or the Debugger. Entering the Debugger is usually more useful, for there you can examine the stack, return values, and take other steps in addition to evaluating forms.

You can use two general kinds of breakpoints:

- You can edit into a definition a call to **zl:dbg** (with no arguments) or to **zl:break**. The advantage of this kind of breakpoint is that, as with stepping, you can interrupt execution within the function. The disadvantage is that you have to edit and recompile the definition to insert and remove the breakpoint. If you redefine the function after inserting the breakpoint, the breakpoint might be lost.

- You can use **breakon** or one of the error or break options to **trace**. These features create *encapsulations*, functions that contain the *basic definitions* of the functions to which you want to add breakpoints. For more on encapsulations: See the section "Encapsulations". The advantage of this kind of breakpoint is that when you recompile or otherwise redefine the function, only the basic defi-

nition 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.

Command and Function Summary

(**zl:dbg** *process*) Enters the Debugger in *process*. With an argument of **t**, finds a process that has sent an error notification. With no argument, enters the Debugger as if an error had occurred in the current process.

(**zl:break** *tag conditional-form*)
Enters a Lisp breakpoint loop (identified as "breakpoint *tag*") if *conditional-form* is not **nil** or is not supplied.

(**breakon** *function-spec conditional-form*)
Passes control to the Debugger on entering *function-spec* if *conditional-form* is not **nil** or is not supplied. With no arguments, returns a list of functions with breakpoints specified by **breakon**.

(**unbreakon** *function-spec conditional-form*)
Turns off the breakpoint condition specified by *conditional-form* for *function-spec*. If *conditional-form* is not supplied, turns off all breakpoints specified by **breakon** for *function-spec*. With no arguments, turns off all breakpoints specified by **breakon** for all functions.

[Error] (from a trace menu)
Passes control to the Debugger on entering a function being traced.

[Cond break before] (from a trace menu)
Prompts for a predicate. Displays trace entry information and enters a Lisp breakpoint loop on entering a function being traced if the predicate is not **nil**.

[Cond break after] (from a trace menu)
Prompts for a predicate. Displays trace exit information and enters a Lisp breakpoint loop on exiting from a function being traced if the predicate is not **nil**.

(trace (:function *function-spec* **:error))**
Passes control to the Debugger on entering a function being traced. If *function-spec* is a symbol, the keyword **:function** is unnecessary.

(trace (:function *function-spec* **:break** *predicate*))
Prints trace entry information and, if the value of *predicate* is not **nil**, enters a Lisp break loop on entering the function. If *function-spec* is a symbol, the keyword **:function** is unnecessary.

(trace (:function *function-spec* **:exitbreak** *predicate*))
Prints trace exit information and, if the value of *predicate* is not **nil**, enters a Lisp break loop on exiting from the function. If *function-spec* is a symbol, the keyword **:function** is unnecessary.

Macro Expansion

Sometimes a program bug appears to stem from unexpected behavior by a macro. Seeing how a macro form expands can help find the bug. To be sure that a macro does what you want it to, you might also want to create and expand a macro form soon after defining the macro and compiling the definition.

You can expand a macro form in a Zmacs buffer using Macro Expand Expression (`c-sh-M`). This command expands the form following point, but not any macro forms within it. To expand all subforms, use Macro Expand Expression All (`m-sh-M`). Without a numeric argument, these commands type their results in the typeout window; with a numeric argument, the commands pretty-print their results in the buffer immediately after the expression. You can also expand macro forms with **mexp**, which enters a loop to read and expand one form after another.

Command and Function Summary

Macro Expand Expression (`c-sh-M`)
Expands the macro form following point. Does not expand subforms within the form. Without a numeric argument, types result in the typeout window; with a numeric argument, pretty-prints the result in the buffer immediately after the expression.

Macro Expand Expression All (`m-sh-M`)
Expands the macro form following point and all subforms within the form. Without a numeric argument, types result in the typeout window; with a numeric argument, pretty-prints the result in the buffer immediately after the expression.

(mexp)
Enters a loop: prompts for a macro form to expand, expands it, and prompts for another macro form. Exits from the loop on `END`.

Using the Inspector

The Inspector is a window-based tool that combines the **describe** and **disassemble** functions. Invoke it with **inspect**, `SELECT I`, or [Inspect] from the System menu. If you use **inspect**, the Inspector is not a separate activity from the Lisp Listener in which you invoke it. In that case you cannot use `SELECT L` to return to the Lisp Listener; you must click on [Exit] or [Return] in the Inspector menu.

The Inspector displays information about an object and lets you modify the object. It displays information for the last object inspected in the bottom window. It displays information for the two previous objects in the windows above the bottom one. It maintains a mouse-sensitive listing of all inspected objects in the history window. These are some of its useful features:

- The information the Inspector displays depends on the object's type. For a symbol, it displays a representation of the value, function, property list, and package. For a symbol's flavor property, it displays information about instance variables, component and dependent flavors, the message handler, init keywords, and the flavor property list. For a compiled function, it displays the disassembled assembly-language code that represents the compiler output.

- The Inspector is especially useful for examining data structures. It displays the names and values of the slots of structures and, unlike **describe**, the elements of (one-dimensional) arrays. For instances of flavors, the Inspector displays the names and values of instance variables.

- Within each display, most representations of objects are mouse sensitive. If you click on an object representation, you inspect that object. For example, you can inspect elements of lists. If an element of an array is itself an array, you can inspect the second array. In this way you can follow long paths in data structures.

- You can change a value by using the [Modify] option in the Inspector's menu. You can return a value when you exit the Inspector by clicking on [Return].

For more on the Inspector: See the section "The Inspector".

Command and Function Summary

(inspect *object***)** Selects an Inspector window in which to inspect *object*.

`SELECT I` Selects an Inspector window.

[Inspect] (from the System menu)
Selects an Inspector window.

(disassemble *function***)**
Prints a representation of the assembly-language instructions for a compiled function.

Disassemble (M-X) Prompts for the name of a compiled function and displays a representation of the function's assembly-language instructions.