

**Medley Interlisp:
Interactive Programming Tools
(derived from Interlisp-D)**

**by
Stephen H. Kaisler, D.Sc**

**Version 1.0
November 2021**

Table of Contents

Table of Contents	2
List of Figures	9
List of Tables	12
Introduction.....	14
1. A Display-Oriented Text Editor.....	16
1.1 Using TEdit.....	16
1.1.1 Text Selection.....	19
1.1.2 Line Bar Selection.....	20
1.1.3 Using the Control Key.....	20
1.1.4 Using the Shift Key	21
1.1.5 Moving Text.....	21
1.1.6 Editing Operations.....	21
1.1.7 TEdit Command Menu	22
1.1.8 Evaluating an Expression	37
1.2 Invoking TEdit.....	38
1.2.1 Editing Files	39
1.2.2 Editing Display Streams.....	39
1.2.3 TEdit Control Properties	40
1.2.3.1 Text Font	40
1.3 TEdit Data Structures	45
1.3.1 The Text Stream.....	46
1.3.2 The Text Object.....	47
1.3.3 The Selection Object	54
1.3.4 The Line Descriptor Object.....	56
1.3.5 The Piece Object	59
1.3.6 The THISLINE Object	60
1.3.7 The Line Cache Object.....	61
1.3.8 The Character Looks Object.....	62
1.3.9 The Format Specification Object	63
1.4 TEdit User Interface Functions.....	65
1.4.1 Opening a Text Stream.....	65
1.4.2 Creating a Text Stream.....	66

1.4.2 Making a Selection in a Text Stream	67
1.4.3 Getting the Current Selection	69
1.4.4 Showing a Selection	69
1.4.5 Inserting into a Text Stream	71
1.4.6 Deleting Text from a Stream	73
1.4.7 Finding Text in a Text Stream.....	74
1.4.8 Generating Hardcopy of Text.....	76
1.4.9 Changing the Looks of Selected Characters.....	77
1.4.10 Getting the Character Looks of a Selection.....	81
1.4.11 Copying Character Looks.....	82
1.4.12 Quitting TEdit	84
1.5 TEdit System Variables	84
1.5.1 Pending Deletions	84
1.5.2 Default Format Specification	84
1.5.3 The Current Selection.....	85
1.5.4The Current Shift-Selection.....	85
1.5.5 The Current Move Selection	86
1.5.6 The Current Read Table	86
1.5.7 The Word Boundary Read Table.....	86
1.5.8 TEdit Default Properties.....	86
2. Display-Oriented Structure Editor	88
2.1 Invoking DEdit	88
2.1.1 Editing Functions	89
2.1.2 Editing Variable Values	90
2.1.3 Editing a Property List	92
2.1.4 Editing File Commands.....	93
2.1.5 The Main DEdit Interface	95
2.2 DEdit Operation.....	96
2.2.1 The Selection Phase	97
2.2.2 The Execute Phase	100
2.2.3 Command Menu.....	101
2.3 DEdit Commands	103
2.3.1 After	103
2.3.2 Before	103
2.3.3 Delete	103

2.3.4 Replace	104
2.3.5 Switch.....	104
2.3.6 Parentheses Insertion.....	104
2.3.7 Parenthesis Removal	105
2.3.8 Undoing the Previous Command	105
2.3.9 Searching for Structure.....	106
2.3.10 Swapping Selections	106
2.3.11 Reprinting a Selection	106
2.3.12 Editing a Structure.....	107
2.3.13 Executing Arbitrary Editor Commands.....	107
2.3.14 Inserting a Break Around an Expression.....	108
2.3.15 Evaluating an Expression	108
2.3.16 Exiting from DEdit.....	108
2.3.17 DEdit Command List	108
Chapter Three.....	111
The Typein Editor (TTYIN)	111
3.1 Using the Mouse with TTYIN	112
3.1.1 Secondary Mouse Operations.....	112
3.2 Invoking TTYIN.....	113
3.2.1 Prompt Characters	114
3.2.2 Spelling Lists.....	116
3.2.3 Help Facility.....	117
3.2.4 The Options List.....	118
3.2.5 Echoing Input to a File	126
3.2.6 Tabbing Specifications	127
3.2.7 Preloading the Input Buffer.....	128
3.2.8 Read Table Mediation	128
3.3 TTYIN Editing Commands	129
3.4 TTYIN Macros	132
3.4.1 The ED Macro	132
3.4.2 The EE Macro	132
3.4.3 The BUF macro.....	133
3.4.4 The TV Macro.....	133
3.4.5 The FIX Macro.....	134
3.5 The ?= Handler	134

3.5.1	User Handling of ?=	134
3.5.2	Reading Intermediate Arguments.....	135
3.5.3	Printing Arguments for ?=.....	137
3.5.4	Enabling ?= Handling.....	138
3.6	TTYIN Utility Functions	139
3.6.1	Calling TTYIN as an Editor	139
3.6.2	Setting the TTYIN Window	142
3.6.3	Creating a TTYIN Scratch File	143
3.7	TTYIN Variables	144
3.7.1	Automatic Window Closing.....	144
3.7.2	Allowing Typeahead	144
3.7.3	Providing Alternative Completions.....	145
3.7.4	Showing Matching Parentheses.....	145
3.7.5	Errorset Protection.....	145
3.7.6	Enabling Escape Completion	145
3.7.7	Caching the TTYIN Edit Window	146
3.7.8	Default Printing Function.....	147
3.7.9	Handling Comments.....	147
3.8	TTYIN READ Macros	147
3.9	Special Responses.....	149
4.	Display Oriented Tools	151
4.1	The Interactive Display-Oriented Break Package	151
4.1.1	Invoking the Display Oriented Break Package	152
4.1.2	Enabling the Display-Oriented Break Package	157
4.1.3	Display-Oriented Break Package System Variables ...	158
4.2	The Inspector	161
4.2.1	Invoking the Inspector.....	162
4.2.2	Inspect Windows	170
4.2.3	Creating and Manipulating Inspector Windows.....	172
4.2.4	Inspect Commands	190
4.2.5	Interacting with Break Windows.....	191
4.2.6	Inspector Variables.....	192
4.2.7	Inspector Macros	194
4.3	The Grapher Utility	196
4.3.1	The Structure of a Graph.....	197

4.3.2	The Structure of a Graph Node	203
4.3.3	The Structure of a Link Description	206
4.3.4	Creating a Node.....	207
4.3.5	Displaying a Graph.....	211
4.3.6	Laying Out a Graph.....	218
4.3.7	Editing a Graph	230
4.3.8	Laying Out a Forest.....	232
4.3.9	Laying Out an S-Expression.....	233
4.3.10	Creating an Image Object for a Graph	235
4.3.11	Determining the Minimal Graph Region.....	236
4.3.12	Inverting a Graph Node.....	237
4.3.13	Resetting the Node Border	238
4.3.14	Resetting the Node's Label Shade	239
4.3.15	Dumping a Graph to a File	240
4.3.16	Reading a Graph from a File	242
5.	Graphics	245
5.1	Basic Concepts	245
5.1.1	Brushes	245
5.1.2	Operations	257
5.1.3	Dashing	257
5.2	Lines and Curves	258
5.2.1	Drawing a Line.....	258
5.2.1.1	From the Current Position	258
5.2.2	Drawing Curves.....	265
5.2.3	Drawing a Gray Box	268
5.3	Rectangles.....	270
5.3.1	Creating a Rectangle	271
5.3.2	Rectangle Functions	276
5.3.3	Rectangle Manipulation Functions.....	279
5.4	Closed Polygons	281
5.4.1	Drawing Circles.....	281
5.4.2	Drawing Ellipses	283
5.5	Filling Objects with Texture.....	284
5.5.1	Filling Polygons	284
5.5.2	Filling a Circle.....	288

6. Process Management	290
6.1 Process Concepts	290
6.1.1 The Structure of a Process.....	291
6.1.2 The TTY Process.....	296
6.1.3 Handling the Mouse	303
6.1.4 Handling Interrupts	305
6.1.5 Enabling the Process World	307
6.1.6 The Process Status Window.....	308
6.2 Creating and Destroying Processes	310
6.2.1 Creating a Process	310
6.2.2 Killing a Process.....	314
6.3 Process Properties.....	314
6.3.1 Getting and Setting Process Properties	315
6.3.2 Process Name	316
6.3.3 The Process Body.....	316
6.3.4 The Restart Flag	317
6.3.5 A Restart Form	317
6.3.6 Processing Before an Exit	318
6.3.7 Processing After an Exit.....	318
6.3.8 An Information Hook	318
6.3.9 Handling the TTY Display Stream.....	319
6.4 Process Management Functions	319
6.4.1 Testing for an Active Process.....	319
6.4.2 Testing for a Deleted Process.....	320
6.4.3 Testing for a Finished Process.....	321
6.4.4 Finding a Process Handle.....	322
6.4.5 Restarting a Process	323
6.4.6 Returning a Value from a Process	324
6.4.7 Mapping Across Processes.....	325
6.5 Process Control Functions	327
6.5.1 Suspending a Process	327
6.5.2 Awakenng a Process	327
6.5.3 Blocking a Process	328
6.5.4 Dismissing a Process.....	329
6.5.5 Evaluating Expressions in a Processes' Context.....	330

6.6 Interprocess Communication	333
6.6.1 Creating an Event	334
6.6.2 Awaiting an Event	334
6.6.3 Signalling Completion of an Event	335
6.7 Monitors: Sharing Data Structures	336
6.7.1 Creating a Monitor	337
6.7.3 Evaluating Expressions under a Monitor Lock	338
6.7.4 Awaiting a Monitor Event	339
6.7.5 Obtaining a Monitor Lock	341
6.7.6 Releasing a Monitor Lock	342
Index	343

List of Figures

- 1-1. TEdit Editing Window
 - 1-2. Normal Editing Cursor
 - 1-3. Line Bar Cursor
 - 1-4. TEdit Command Menu
 - 1-5. Find Command Result
 - 1-6. Expanded TEdit Command Menu
 - 1-7. The Character Looks Menu
 - 1-8. Using the Character Looks Menu
 - 1-9. Using the Character Looks Menu
 - 1-10. The Paragraph Looks Menu
 - 1-11. The Page Layout Menu
 - 1-12. Evaluating an Expression
 - 1-13. Example of Initial Text Selection
 - 1-14. TEdit Text Object Example
 - 1-15. TEdit Selection Object Example
 - 1-16. TEdit Line Descriptor Object Example
 - 1-17. TEdit Piece Object Example
 - 1-18. TEdit THISLINE Object Example
 - 1-19. Line Cache Object Example
 - 1-20. TEDIT.SHOWSEL Example
 - 1-21. TEDIT.INSERT Example
 - 1-22. TEDIT.INSERT with NIL selection
 - 1-23. TEDIT.HARDCOPY Example
 - 1-24. Example 1 of TEDIT.LOOKS
 - 1-25. Example 2 of TEDIT.LOOKS
 - 1-26. Example of TEDIT.COPY.LOOKS
-
- 2-1. Editing a Function Definition
 - 2-2. Editing a Variable Value
 - 2-3. Editing of a Property List
 - 2-4. Editing File Package Commands
 - 2-5. Selecting an Item via the LEFT Mouse Button

- 2-6. Selecting a Containing List
- 2-7. Shift-Selection to the Type-in Buffer

- 3-1. Example for LOOK.AT.ARGS
- 3-2. TTYINEDIT Editing a String
- 3-3. TTYINEDIT Editing a Function

- 4-1 The Break Window
- 4-2 Display-Oriented Break Package Command Menu
- 4-3 Inspecting a Stack Frame
- 4-4 Stack Frame Inspection Commands
- 4-5 Setting EMODE
- 4-6 Operations on a Stack Frame
- 4-7 Inspection of an IMAGEOBJ Record
- 4-8 Example of INSPECTCODE
- 4-9 Inspecting in a Break Window
- 4-10 Inspecting via EDITV
- 4-11 INSPECT/ARRAY Example
- 4-12 A Sample Inspect Window
- 4-13 Inspector Window with Property Menu
- 4-14 Inspector Window with Property Value Menu
- 4-15 Inspect Window for a Text Object
- 4-16 Inspect Window Properties
- 4-17 FETCHFN Example
- 4-18. STOREFN Example
- 4-19 INSPECTW.REPLACE Example
- 4-20 INSPECTW.SELECTITEM Example
- 4-21 Example of SHOWGRAPH
- 4-22. Example of Add Node
- 4-23. Adding a Link Example
- 4-24- Examples of Enlarging and Reducing Labels
- 4-25 A Graph laid out in COMPACT Format
- 4-26. A graph laid out in FAST Format
- 4-27. A graph laid out in LATTICE Format
- 4-28. A Graph laid out in REVERSE Format

4-29 Example of a COMPACT HORIZONTAL Graph

4-30. Example of a COMPACT VERTICAL Graph

5-1. An Example of a ROUND Brush with dashing

5-2. Example of a ROUND Brush

5-3. Example of a SQUARE Brush

5-4. Menu of Brush Shades

5-5. Tailoring a 4x4 Shade

5-6. The Tailored Shade

5-7. A Checkered Brush

5-8. A DRAWTO Example

5-9. Example of RELDRAWTO

5-10. Example of DRAWLINE

5-11. An Example of DRAWBETWEEN

5-12. A DRAWCURVE Example

5-13. Another DRAWCURVE Example

5-14. Example of DRAWGRAYBOX

5-15- RECTANGLE Example

5-16. DRAWCIRCLE Example

5-17. DRAWELLIPSE Example

5-18. An Example Using FILLPOLYGON

5-19. Another Example of FILLPOLYGON

5-20. FILLCIRCLE Example

6.1 Depiction of HELP Interrupt Menu

6.2 PROCESS DEMO WINDOW

List of Tables

- 1-1. Effect of Mouse Buttons in Text Region
- 1-2. Effect of Mouse Buttons in the Line Bar
- 1-3. Expanded Menu Items
- 1-4. Text Stream Record Structure
- 1-5. Text Object Record Structure
- 1-6. The Selection Object
- 1-7. The Line Descriptor Object
- 1-8. The Piece Object
- 1-9. Structure of the THISLINE Object
- 1-10. Structure of the Line Cache Object
- 1-11. The Character Looks Object
- 1-12. Structure of the Format Specification
- 1-13. TEDIT.INSERT Options
- 1-14. NEWLOOKS Property Values

- 2-1. Mouse Button Effects
- 2-2. Editing Commands in Type-In Buffer
- 2-3. Parenthesis Insertion Subcommands
- 2-4. Parenthesis Removal Subcommands
- 2-5. Undo Subcommands
- 2-6. Swap Subcommands
- 2-7. Exit Subcommands

- 3-1. TTYIN Mouse Button Usage
- 3-2. TTYIN Secondary Mouse Button Usage
- 3-3. Selection Functions
- 3-4. Reading Commands
- 3-5. TTYIN Editing Commands
- 3-6. ?=FN Values
- 3-7. ?= Handling Values
- 3-8. TTYIN Read Macro Codes

3-9. TTYIN Read Macro Response

4-1. AUTOBACKTRACEFLG Values

4-2. Common Menu Entries

4-3. Inspect Window Middle Mouse Button Commands

4-4. Graph Record Fields

4-5. SIDESFLG Interpretation

4-6. SIDESFLG Interpretation

4-7. GRAPHNODE Fields

4-8. Graphnode Border Values

4-9. GraphNode Shade Values

4-10. Graph Editing Functions

4-11. Basic Formats

4-12. Graph Orientation Values

4-13. MARK Format Values

4-14. Node Distance Criteria

4-15. Boxing Values

6-1. Process Structure

6-2. Process Status Window Commands

6-3. RESTARTABLE Values

6-4. AFTEREXIT Values

6-5. Event Structure

6-6. Monitor Lock Structure

Introduction

This volume focuses on a set of tools for the interactive programming interface for Medley Interlisp. I tried to select the tools that I felt were extremely useful to the Interlisp user. Some tools were omitted due to space limitations. The decisions are solely mine.

Chapter 1 describes the Display-oriented Text Editor, TEdit. TEdit is a modeless text editor that is used to create large text files. It incorporates many of the powerful ideas that were first available in the Bravo Text Editor on the Alto. An essential idea that is supported by TEdit is that each editing session is an independent process. Thus, you may edit multiple documents - perhaps using a "cut and paste" method of composition. TEdit allows you to adjust the individual appearance of characters, string of characters, paragraphs of text, and even the entire page layout. Most of TEdit is menu-driven which makes it easy to modify text appearances using the "point and click" paradigm.

Chapter 2 describes the Display Oriented Structure Editor, DEdit. DEdit "understands" Interlisp data and function structures. I have found that the best way to learn DEdit is to play with it. Rather than reading a manual, one learns to use DEdit by interactive experimentation. Note that this paradigm has been successfully parlayed by Apple Computer in its Macintosh family.

Chapter 3 describes the Typein Editor, TTYIN. TTYIN provides a very flexible editing environment for reading input that is typed in by the user. Many command interfaces have been built using the TTYIN subsystem. Several of Interlisp's subsystems, including the Inspector, use portions of the TTYIN subsystem to interact with the user.

INTRODUCTION

Chapter 4 describes an enhanced Break Package that uses the display capabilities to debug and trace programs. Basically, it adds a window interface to the Break Package described in *Interlisp: The Language and Its Usage* by SHKaisler. However, I have found that this interface greatly improves one's ability to debug programs - particularly when you can have several windows open on different functions.

An integral Inspector allows the user to view code in one window, the stack in another, and the output of a running program in another while attempting to debug a program. The Inspector is oriented to examining all the data structures supported by Interlisp. In addition, it contains a programmable interface which allows you to construct INSPECTWs on your own data structures.

Finally, this chapter describes the Grapher Utility which allows you to draw directed graphs and lattices from data structures. I believe that this program is well worth the time spent to master its intricacies because of the visual power it gives the user over complex environments.

Chapter 5 describes the Graphics Package incorporated into Medley Interlisp. Most of the window management routines are built on top of this package. You can create complex interactive graphics packages by combining the capabilities in this package with the image streams subsystem.

Chapter 6 describes the Process Management Package, a package of functions that allows the user to initiate and control multiple asynchronously executing processes. Essential to the multitasking environment are events (somewhat like semaphores) and monitors which are based on Hoare's ideas.

1. A Display-Oriented Text Editor

The Text Editor (TEdit) is a display-oriented, modeless text editor based on principles similar to those employed to implement DEDIT. It can be used to create and edit large text files. Within a text file, you may apply a variety of formatting commands to affect the appearance of characters and sequence of characters, paragraphs, and page layouts. In addition, using image streams, you are able to insert image objects into a TEdit file in order to create system documentation.

Because TEdit represents a considerable piece of software, I have not tried to exhaustively describe all of its features. Some of the more esoteric functions have been omitted in the interest of space. You may refer to the description of TEdit in the Lisp Library Manual for a description of the omitted features.

1.1 Using TEdit

When TEdit is activated, a window is opened with the appropriate data structure displayed in its editing pane. If no object is to be edited, the TEdit window is opened with a clear editing pane. Figure 1-1 depicts a TEdit window.

TEdit

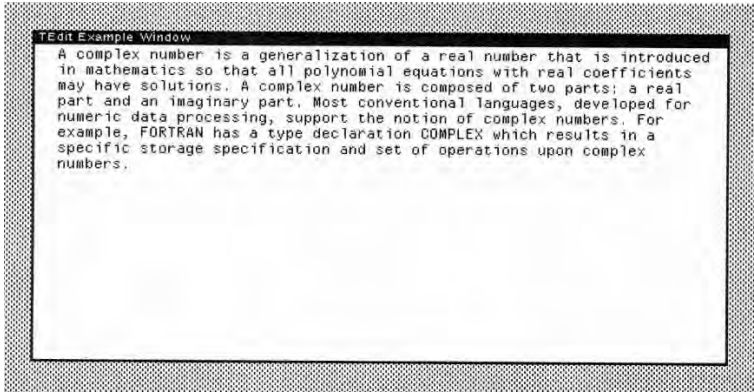


Figure 1.1. TEdit Editing Window

TEdit operates on pieces of text which you select via the mouse. The selected text is highlighted (e.g., inverted). A set of commands is available through the middle mouse button.

The *editing pane* has two regions which are mouse-sensitive. The larger region which contains the displayed text causes the cursor to take the normal up-and-left pointing arrow shape. Inside the left edge of the editing pane is the *line bar*. When the cursor moves into the line bar, the cursor takes the shape of an up-and-right pointing arrow. Figures 1-2 and 1-3 depict these two shapes.

Above the Editing pane is the *title bar* which identifies the data structure that is currently being edited. Placing the cursor in the title bar and pressing the middle mouse button will cause the TEdit Command Menu to pop up.

Above the title bar is the TEdit *prompt pane*. When you activate a TEdit function, a prompt message may be displayed here. If data are required by TEdit to perform the command, the data are entered in this pane.

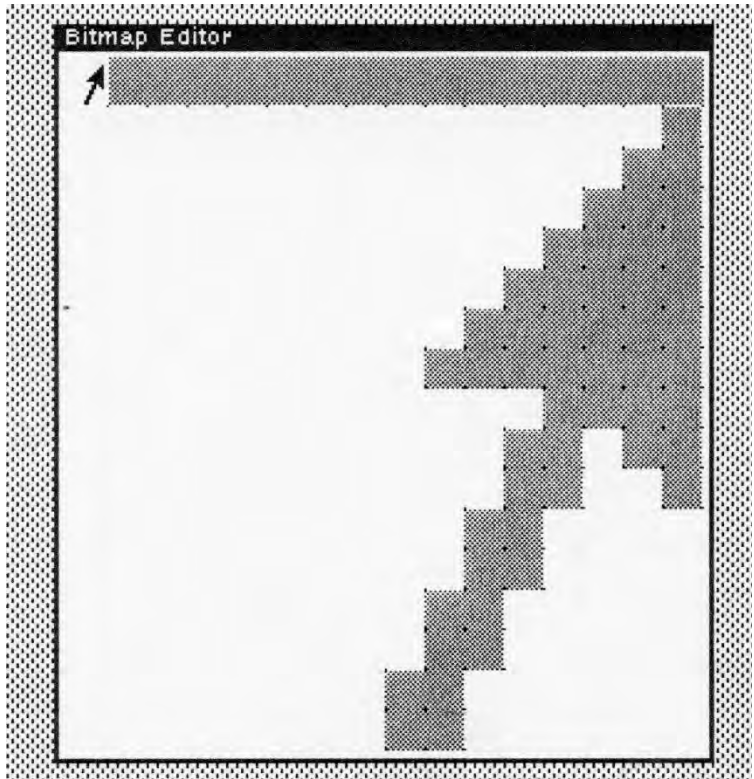


Figure 1-2. Normal Editing Cursor

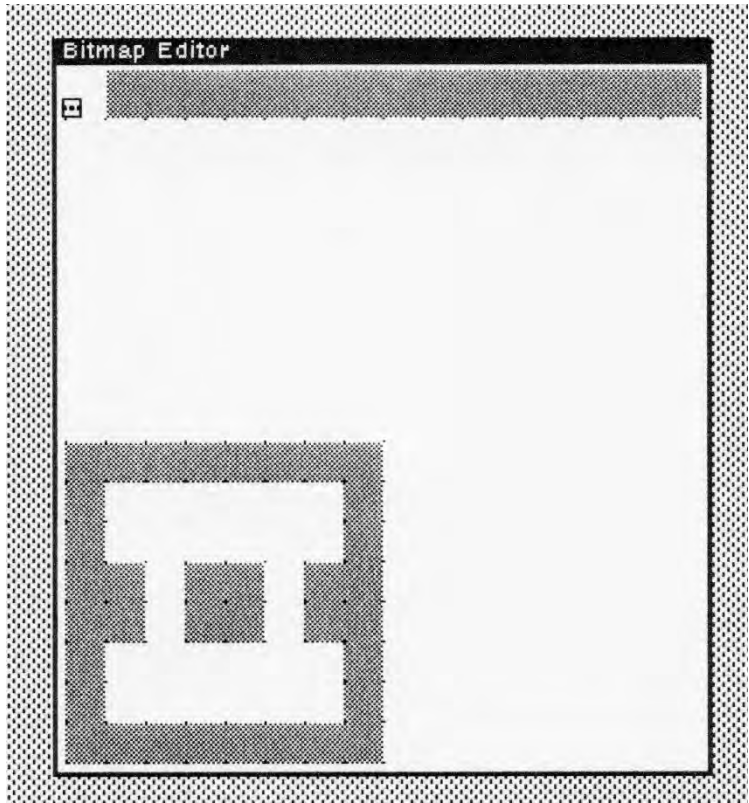


Figure 1-3. Line Bar Cursor

The location of the cursor determines the type of selection of text that is made by TEdit.

1.1.1 Text Selection

To change text that appears in the edit pane, you must first select it and then give a command to affect the change. While the cursor is

in text region of the edit pane, the mouse buttons have the function presented in Table 1-1:

Table 1-1. Effect of Mouse Buttons in Text Region

Button	Effect
LEFT	Selects a character in the text.
MIDDLE	Selects a word in the text.
RIGHT	Extends the selection to include additional items of the same type.

When you have selected text, it is highlighted by underlining or reverse video.

1.1.2 Line Bar Selection

While the cursor is in the line bar region of the edit pane, the mouse buttons have the functions presented in Table 1-2.:

Table 1-2. Effect of Mouse Buttons in the Line Bar

Button	Effect
LEFT	Selects the line corresponding to the hot spot of the cursor.
MIDDLE	Selects the paragraph which the cursor is next to.
RIGHT	Extends the selection to include additional items of the same type.

1.1.3 Using the Control Key

If you hold the control key (CTRL) down while selecting text, the text is displayed as white-on-black. When the CTRL key is released, the selected text will be deleted from the text region. You can abort the delete operation by pressing any mouse button and then

TEdit

releasing the CTRL key. Of course, you must also release the mouse button.

1.1.4 Using the Shift Key

If you hold the shift key down while making a selection, it indicates the source field for a *copy* operation. The selected text will be underlined with a dashed line. When you release the SHIFT key, the underlined text will be copied to the current location of the caret. Copying will work between TEdit windows. You can abort a copy operation by pressing any mouse button and then releasing the SHIFT key. Of course, you must also release the mouse button.

1.1.5 Moving Text

You may move text from one location to another within the same or a different edit window. To do so, you must hold the CTRL and SHIFT keys down while selecting the text to be moved. The text to be moved will be displayed in reverse video. When the CTRL and SHIFT keys are released, the text will be moved to the current location of the caret.

You may abort a move operation by pressing any mouse button and then releasing the CTRL and SHIFT keys. Of course, you must also release the mouse button.

1.1.6 Editing Operations

TEdit provides you with several editing operations through the use of keys on the keyboard. You first make a selection with the mouse and then press the appropriate key(s).

1.1.6.1 Inserting Text

TEdit

A caret indicates the current location, called the *type-in point*, in the text stream for the insertion of new text. New text may be typed from the keyboard or copied from another location within the same or a different edit window. There is only one type-in point active at a time. You must remember, when pointing with the cursor at a location in the text, to press the left mouse button to let TEdit know about the new type-in point.

The BACKSPACE key is used to delete characters one key at a time. In lieu of the BACKSPACE key, you may also type CTRL-A. To delete an entire word, type CTRL-W.

1.1.6.2 Deleting Text

You may delete the currently selected text by pressing the DEL key. TEdit automatically closes the gap left by the deleted text.

1.1.6.3 Undoing an Edit Operation

You may undo the most recent edit operation by pressing the UNDO key. The Undo operation is itself undo-able. You cannot back up more than one edit operation. The location of the UNDO key depends on the type of workstation and keyboard that you have.

1.1.6.4 Redoing an Edit Operation

You may redo an edit operation by pressing the ESCAPE key (the ESC key)or the Redo key (on the Xerox 1186, this is called the Again key). This causes the most recent edit operation to be redone on the current selection. Thus, you may insert text, move the caret to a new location, hit the Redo key, and see the same text inserted in the new location.

1.1.7 TEdit Command Menu

TEdit

You can activate the TEdit Command Menu by moving the cursor to the edit window's title pane and pressing the middle mouse button. A pop-up menu displaying the TEdit commands will appear at the current location of the cursor. The following sections describe the commands and their effect. Figure 1-4 depicts the TEdit Command Menu.

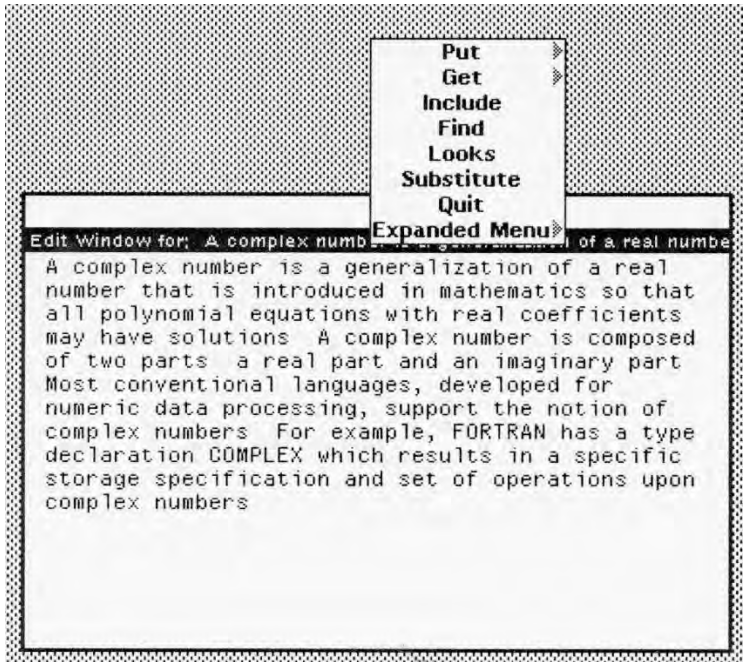


Figure 1-4. TEdit Command Menu

Although the following discussion refers to files, in fact the commands apply to any object which TEdit is capable of editing.

1.1.7.1 Writing a New Version of the File

TEdit

Selecting the ***Put*** command from the TEdit command menu causes a new version of the file to be written to the storage media on which the file is located. TEdit prompts you for a file name, but offers the current file name as a default. Pressing the Return key causes the next version of the file to be written. Alternatively, you may alter the file name and then press the Return key whence a file by the specified name is written on the storage media.

The ***Put*** command has two options which are identified by moving the cursor to the right past the gray triangle. These options are:

- Plain-Text
- Old-Format

When ***Put*** is selected, TEdit prompts you to specify a file name with a message in the TEdit Prompt pane. The form of the message is:

File to Put to: {DSK}<LISPPFILES>EXPORTS.ALL

The cursor is placed after the last letter of the file name. Thus, if you wish to change the file name, you may backspace over the characters that you wish to change and, then, type the new file name.

If you are editing a file, the default file name is the name of the current file. TEdit uses TTYIN to manage the entry of data in this pane.

The *Plain-Text* option strips out editing and format characters such as font information and stores a plain ASCII version of the text on the file.

The *Old-Format* is retained for compatibility with previous versions of TEdit.

TEdit

1.1.7.2 Getting a New File to Edit

Selecting the **Get** command from the TEdit command menu reads a new file into the TEdit window for editing. The previous file is [usnot]us saved. TEdit will prompt you for the name of a file to read in in the prompt window.

The **Get** command provides an optional unformatted read capability. When you drag the cursor to the right of the gray triangle, the option **Unformatted Get** pops up. You may select this option by moving the cursor into the box and releasing the mouse button.

TEdit prompts you for the file to read by displaying the following message in the TEdit Prompt pane:

File to Get:

The cursor is placed one space beyond the ":" to indicate that you should type the name of the file there.

Note that you should type the complete file name, including the directory name. Otherwise, TEdit defaults to your current directory. If the file is located on another host in a network, you must also supply the host name.

1.1.7.3 Including One File in Another File

Selecting the **Include** command from the TEdit command menu allows you to copy the contents of a specified file into the edit window at the current location of the caret. TEdit prompts you for the name of the file to include in the prompt window. The file's contents are copied into the TEdit window at the current location of the caret.

1.1.7.4 Exiting TEdit

Selecting the *Quit* command from the TEdit command menu allows you to exit TEdit without saving the file that you are editing. If you have made changes to the file since you last saved it on the storage media, you will be asked to confirm the immediate exit.

1.1.7.5 Finding a Text String

Selecting the *Find* command from the TEdit command menu allows you to search for a string of text within the file that you are editing. TEdit prompts you for the search string through a query displayed in the prompt window. It then searches for the specified text string from the caret through the end of the file. If the text string is found (e.g., the first instance), TEdit positions the caret to the left of the string. If the text string is not found, TEdit displays the message "not found" in the TEdit prompt pane.

When you select Find in the TEdit menu. You are prompted to enter the text by the message "Text to Find:" in the TEdit prompt pane. You enter "FORTRAN". TEdit displays the message "Searching ... Done" in the prompt pane and places the caret to the left of the text in the edit pane. FORTRAN is underlined as depicted in Figure 1-5.

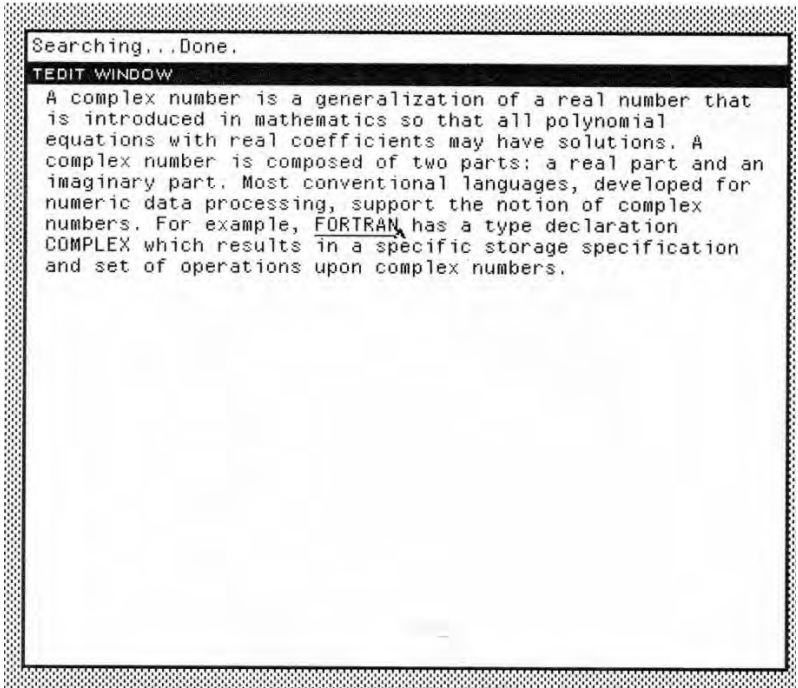


Figure 1-5. Find Command Result

1.1.7.6 Substituting One Text String for Another

Selecting the *Substitute* command from the TEdit command menu allows you to replace one text string by another. You are prompted for the the search string and the replacement string in the prompt window. In the default case, all instances of the search string will be replaced by the replacement string in the current selection.

When you select Substitute, TEdit prompts you to enter the search string in the prompt pane via the message "Search string:". After you enter the search string, it requests a replacement string via the message "Replace string:". After you type the replacement

TEdit

string, TEdit queries you concerning confirmation of each replacement via "Ask before each replace? No". If you press <CR>, TEdit performs a global replacement. Alternatively, you may type "Yes" and TEdit will request confirmation before each replacement.

1.1.7.7 Changing the Appearance of Text

Selecting the *Looks* command from the TEdit command menu allows you to change the appearance of the current selection. The characteristics which may be changed are: the font, the character size, and the face.

Three menus pop up in sequence from which you make selections for the font, character size, and face.

The Font Menu

The Font Menu allows you to select a font from the fonts known to TEdit by searching the font directories. The current fonts that may be displayed are:

- Classic
- Modern
- Terminal
- Titan
- Gacha
- Helvetica
- Times Roman
- Other

The Type Size Menu

The Type Size Menu allows you to determine the type size. TEdit determines the type sizes available for the fonts that it has access to by searching the font directories. In my system the type

sizes are displayed as a menu of 3 columns by 4 rows. However, the number of type sizes varies with the available fonts.

The Face Menu

The Face Menu allows you to change the face of the font that you have selected. The Face menu is composed of four options:

- Bold
- Italic
- Bold Italic
- Regular

You may avoid changing any characteristic by clicking the left mouse button while the cursor is located outside of the corresponding menu.

1.1.7.8 TEdit Expanded Menu

Selecting the *Expanded Menu* command from the TEdit command menu causes TEdit to display an Expanded TEdit Menu above the prompt pane. Figure 1-6 depicts the Expanded Menu. The following sections discuss the commands in the Expanded Menu.

TEdit

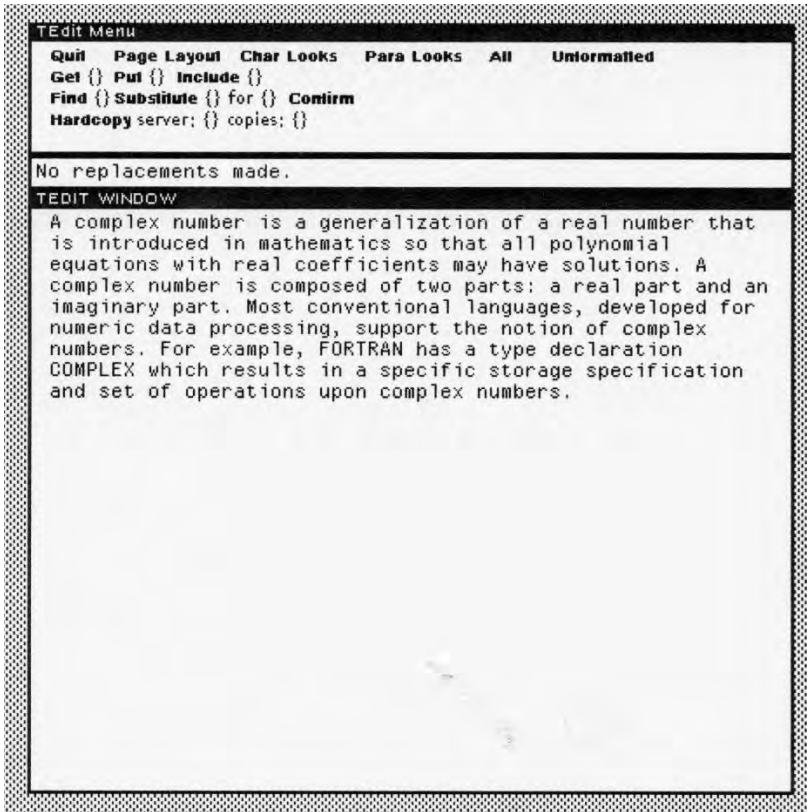


Figure 1-6. Expanded TEdit Command Menu

Note that some of the commands in the primary TEdit menu are replicated here. You may use the Get, Put, Find, and Substitute commands by placing the cursor between the curly brackets and typing the file name or string (respectively) that is the argument for that command.

You use the Confirm entry to confirm an operation.

Table 1-3. Expanded Menu Items

Items	Description
Page Layout, Char Looks, and Para Looks	All cause additional menus to be "stacked" on top of the expanded TEdit menu.
All	This item causes all text in the edit pane to be selected. This allows you to apply a single operation, such as changing the font of the text, at once.
Unformatted	This item treats the text as unformatted. This last command is useful when you are creating text which is to be sent to another computer system.
Hardcopy	This item causes TEdit to print a copy of your file on the default printer. It assumes 1 inch margins around the text of the document.
PRINTERMODE	This item controls the type of printer to which TEdit will send your document for printing.

Using the Expanded Menu entry, you can display three additional menus which allow you to modify the appearance of characters, paragraphs, and pages. For more detailed information, you should consult the TEdit documentation.

1.1.7.9 The Character Looks Menu

The Character Looks Menu is depicted in Figure 1.7. It allows you to modify the properties of a text selection including the font, the face, the size, whether or not it is underlined or overstruck, and whether or not it is sub- or super-scripted. Two examples using the Character Looks Menu is depicted in Figures 1-8 and 1-9.

TEdit

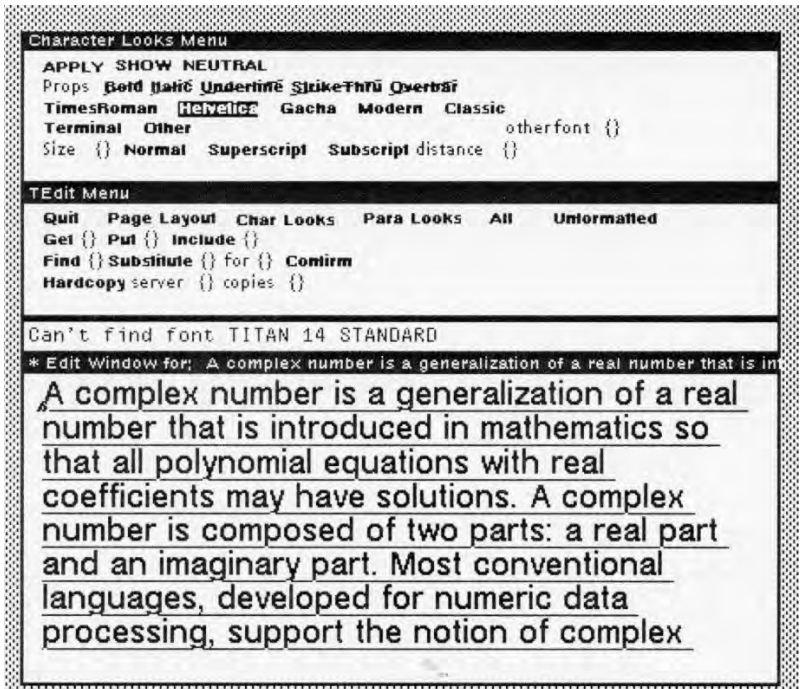


Figure 1-7. The Character Looks Menu

TEdit

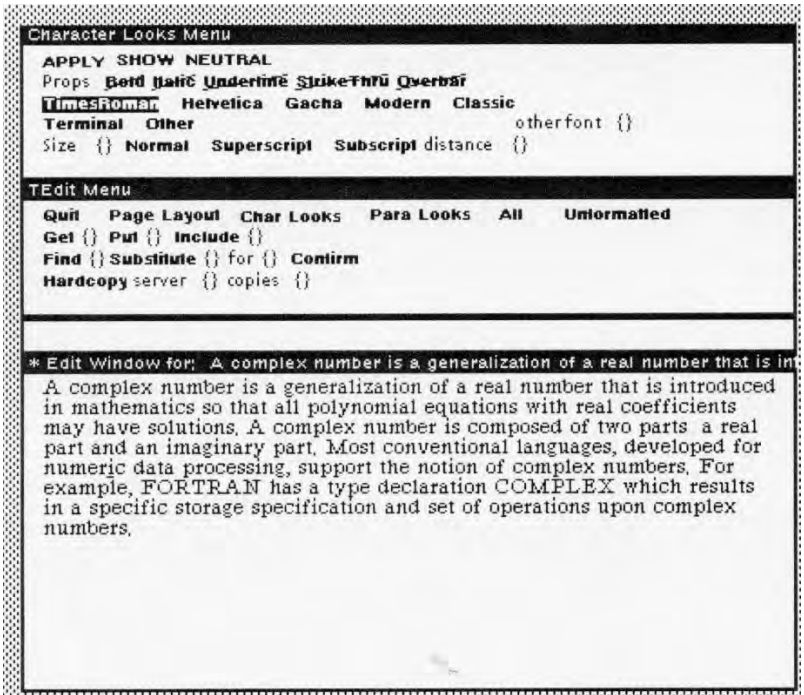
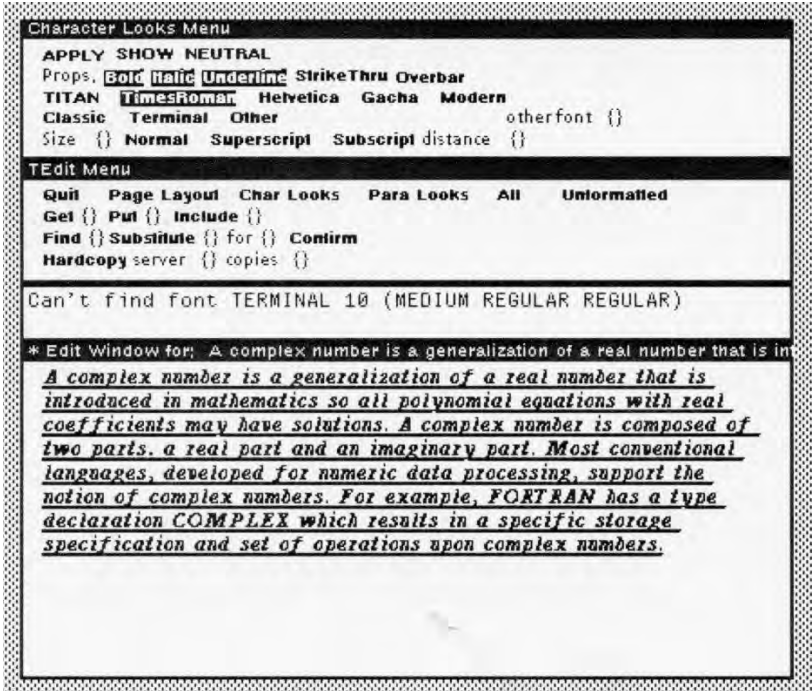


Figure 1-8. Using the Character Looks Menu



Figure

1.1.7.10 Paragraph Looks Menu

The Paragraph Looks Menu is depicted in Figure 1-10. It allows you to modify the appearance of a paragraph. Basically, you can determine the justification of the lines which compose the paragraph. You may also specify tabbing for the paragraph and whether or not a new page should occur before or after the paragraph.

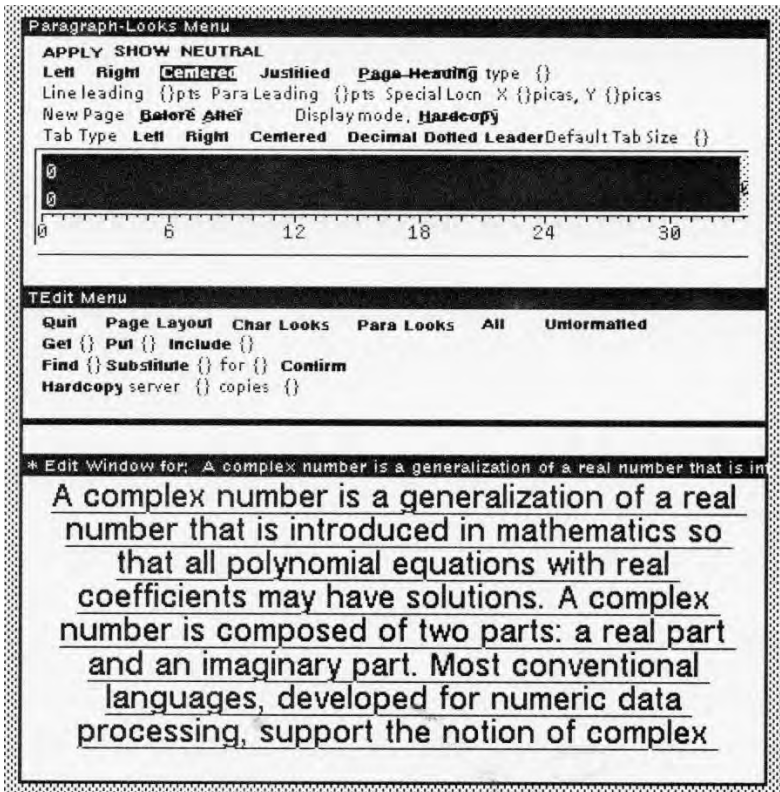
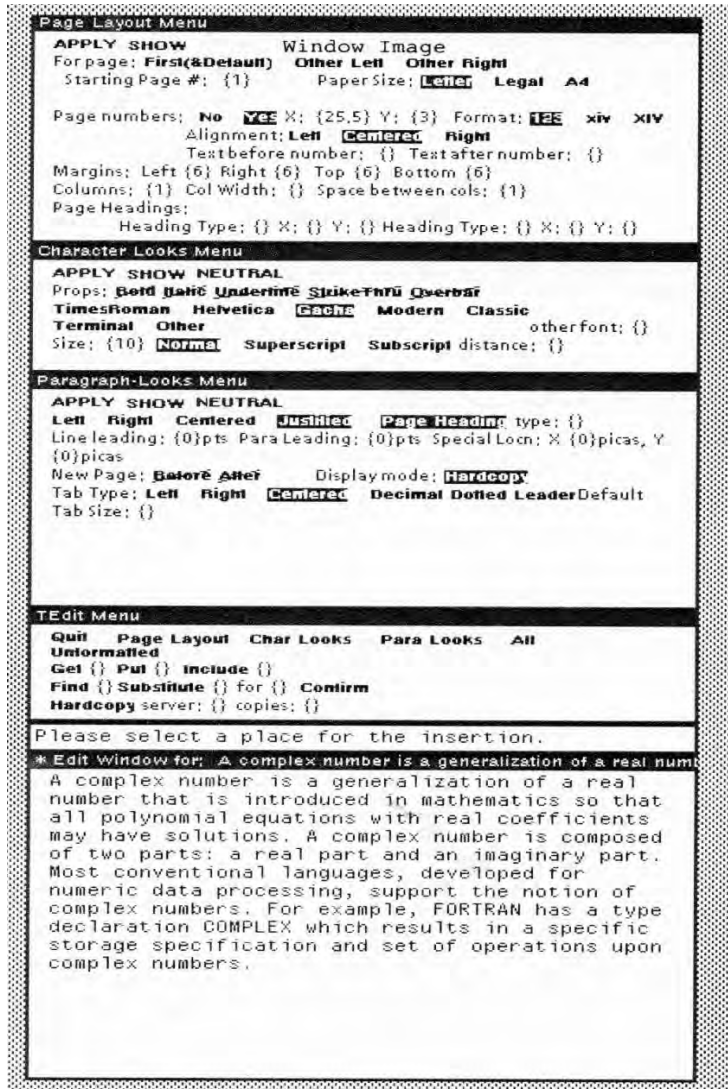


Figure 1-10. The Paragraph Looks Menu

1.1.7.11 Page Layout Menu

The Page Layout Menu is depicted in Figure 1-11. It allows you to specify how individual pages or a sequence of pages in the document will appear. You may specify the left and right margins, the page numbering, the number of header lines for the page, the character looks for the page numbers (including fonts), and the page size. You may also specify whether the image is rotated on the page to landscape mode.

TEdit



Figure

1.1.8 Evaluating an Expression

In some cases, you may want to insert the value of an expression in a TEdit document. By pressing <CTRL>O, you can open a window (see Figure 1-12) in which you can type an expression. The window is created by TTYIN. After you have typed the expression and pressed <CR>, the value of the expression will be inserted at the current location of the caret.

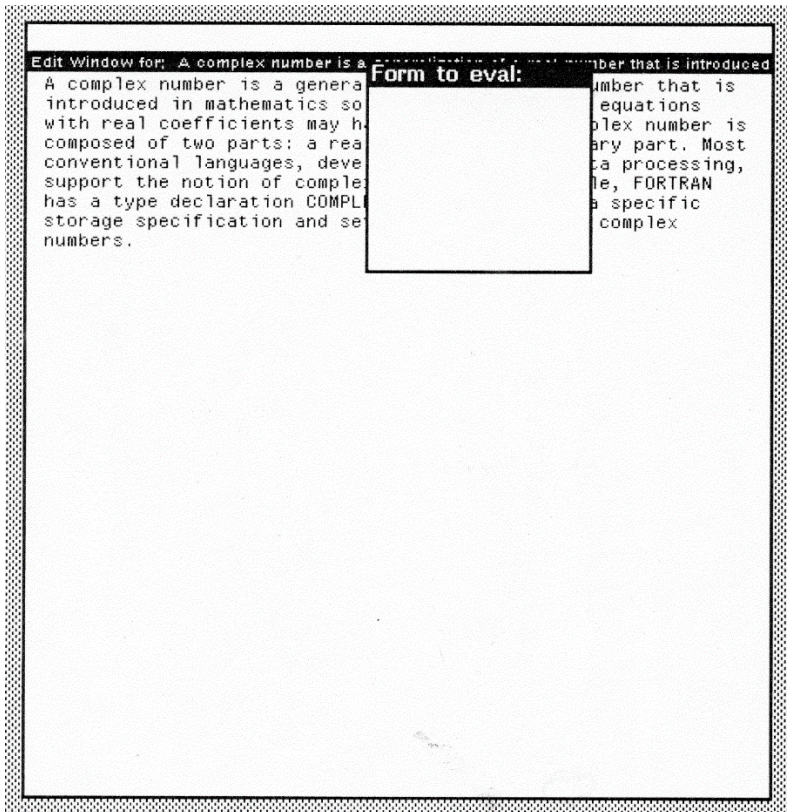


Figure 1-12. Evaluating an Expression

TEdit

1.2 Invoking TEdit

You may start TEdit by either selecting the menu entry in the Background Menu or executing a function. You may invoke TEdit by executing the function of the same name, **TEDIT**, which takes the form:

Function:	TEDIT
# Arguments:	4
Arguments:	1) TEXT, a text specification 2) WINDOW, a window handle 3) DON'TSPAWN, a new process flag 4) PROPS, a property list containing property/value pairs that control the editing session
Value:	A process handle.

TEdit may be invoked on different types of text objects: files, display streams, strings, or arbitrary Interlisp objects. These cases are discussed in Sections 1.2.1 and 1.2.2.

WINDOW specifies a window handle which will be used to display the text for editing. If WINDOW is NIL, you are prompted to create a window. TEdit preserves the title associated with the window, if any, while it is using the window for editing tasks. If the window has no title, TEdit supplies one.

DON'TSPAWN is a flag used to control the spawning of an independent editing process. TEdit will spawn a new process to perform the editing tasks. To prevent the creation of a new process, invoke TEdit with DON'TSPAWN set to T.

TEdit

PROPS is a list of property/value pairs that specify options which control the editing session. The properties are described in Section 1.2.3. Consider the following example:

```
<-(TEDIT myext awindow T)
```

(A complex number is a generalization of a real number that is introduced in Mathematics so that all polynomial equations with real coefficients may have solutions. A complex number is composed of two parts: a real part and an imaginary part. Most conventional languages, developed for numeric data processing, support the notion of complex numbers. For example, FORTRAN has a type declaration COMPLEX which results in a specific storage specification and set of operations upon complex numbers.)

In this case TEDIT did not spawn a process for editing. Thus, TEDIT returned the contents of the window as you edited it. This is useful for typing in information quickly, while having access to sophisticated editing tools. On the other hand, if we spawned a process, the TEDIT would return the process handle as follows:

```
<-(TEDIT mytext awindow)
{PROCESS}#77,110700
```

1.2.1 Editing Files

If TEXT is an atom, it is assumed to be a file name. The contents of the file, if it exists, are read into the window associated with TEdit. For example, to edit the contents of EXPORTS.ALL:

```
<-(TEDIT 'EXPORTS.ALL)
{PROCESS}#71,22300
```

1.2.2 Editing Display Streams

TEdit

TEXT may have a display stream handle as its value. The contents of the display stream are displayed in the TEdit window. Consider the following example:

```
<-(SETQ mystream (OPENTEXTSTREAM mytext))  
{STREAM}#64,126000
```

```
<-(TEDIT mystream)  
{PROCESS}#56,56500
```

Note that we had to open the display stream (in this case a text stream) before its contents could be displayed for editing. Otherwise, you will receive the message "FILE NOT OPEN".

1.2.3 TEdit Control Properties

You may specify values for a number of control properties which allow you to customize your editing session. These properties are discussed in the following sections.

1.2.3.1 Text Font

TEdit will display text using inherent font information. If no font is specified with the text, TEdit uses the font specified by the variable, TEDIT.DEFAULT.FONT, whose initial value is:

```
<-TEDIT.DEFAULT.FONT  
NIL
```

This causes TEdit to use the fonts specified in the system font profile. You may override the default font when you invoke TEdit by specifying a value for the property FONT in PROPS. This property will only be used if the LOOKS property is not specified.

TEdit

1.2.3.2 Exit Procedure

You may specify a function (or list of functions) to be called when you *Quit* TEdit by assigning a value to the property QUITFN in PROPS. This function will be called before TEdit terminates. However, if the function returns the atom DON'T, TEdit does not terminate.

Typically, this function will perform any cleanup operations or formatting prior to closing a document. If any of the functions returns T, you are not asked to confirm the exit quit command - even if the document has unsaved changes.

1.2.3.3 Loop Procedure

You may specify a function to be called each time TEdit runs through the character read loop by assigning a value to the property LOOPFN in PROPS. Typically, you will use this procedure to perform macro substitution as TEdit reads individual characters. Note that you must also consider the impact of the read tables on Tedit's performance.

1.2.3.4 Character Entry Procedure

You may specify a function to be called for each character that is typed in by assigning a value to the property CHARFN in PROPS. This function operates similarly to the one specified for LOOPFN. However, this function applies only to characters which are typed into the edit pane.

1.2.3.5 Text Selection Function

You may specify a function to be called each time a text selection is made in the window using the mouse by assigning a value to the property SELFN in PROPS. This function is useful if

TEdit

you want to protect certain areas of a window from being changed by the user. Using this capability, you can use TEdit to build a forms manager.

1.2.3.6 Text Terminal Table

You may specify a terminal table for displaying characters in alternative ways by assigning a value to the property `TERMTABLE` in `PROPS`.

1.2.3.7 Read-Only Windows

You may specify that the text display window will be a read-only window by including this atom in `PROPS`. In a read-only window, you may only perform shift-select operations (see Section 1.1.4).

Read-Only windows are useful for inspecting original text which should not be deleted.

1.2.3.8 Initial Text Selection

You may specify the text that should be selected upon entry to TEdit by assigning a value to the property `SEL` in `PROPS`. This value may be a selection handle, a character index, or a two element list consisting of a character index and the number of characters to be selected. If the value of this property is the atom `DON'T`, then nothing will be selected initially. This is the default value. Consider the example:

```
<-(TEDIT MYTEXT IOWINDOW NIL '(SEL (23 14)))  
{PROCESS}#77,110000
```

TEdit

which displays the value of MYTEXT in the edit pane. The word "generalization" is underlined in the edit pane to indicate it has been selected (see Figure 1-13).

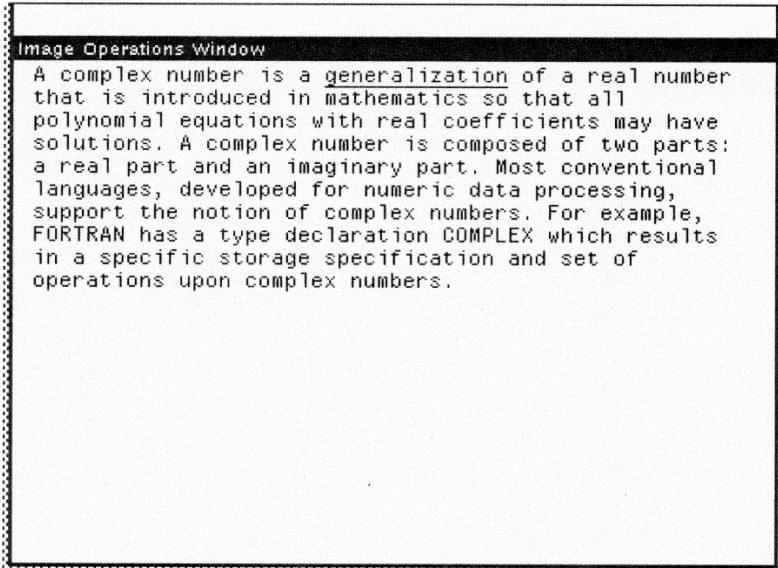


Figure 1-13. Example of Initial Text Selection

1.2.3.9 Middle Button Menu

The middle mouse button, when pressed in the edit window's title region, will display a user-programmable menu. You may specify the menu handle that is to be displayed when you press the middle mouse button by assigning a value to the property MENU in PROPS.

The value may be either a menu handle, whence the menu will be displayed, or a list of menu items. In the latter case, TEdit

TEdit

constructs a menu and displays it when the middle mouse button is pressed.

1.2.3.10 TEdit Cleanup Function

You may specify a function to be called after TEdit has terminated by assigning a value to the property AFTERQUITFN in PROPS. This function will normally be used when you have called TEdit from within a program without creating a new process. Because the edited results are returned by TEdit, you may want to perform additional processing on them before storing them someplace.

1.2.3.11 PROMPTWINDOW Specification

You may specify a window to be used for unscheduled user interactions, e.g., an alternative prompt window by assigning a value to the property PROMPTWINDOW. This window overrides the window normally used by TEdit. If the value is DON'T, the TEdit will use the main prompt window.

1.2.3.12 Alternative Title Menus

You may specify alternative menus to be displayed when the left or middle mouse buttons are pressed in the edit window's title region by assigning a value to the property TITLEMENUFN in PROPS.

1.2.3.13 Character Looks

The value of the property LOOKS specifies the default character looks for characters displayed in the text pane. This value may be a font descriptor, a CHARLOOKS handle, or a property list of character looks properties that is acceptable to TEDIT.LOOKS (see Section 1.4.9).

1.2.3.14 Paragraph Looks

The value of the property PARALOOKS specifies the default paragraph looks to be used for paragraphs in the text being edited. Its value may be either a FMTSPEC handle or a property list of paragraph format properties that is acceptable to TEDIT.PARALOOKS.

1.2.3.15 Caret Looks

The value of CARETLOOKSFN is a function that is called whenever a new caret looks is to be set. If it returns NIL, the looks of the caret will be changed. If it returns a CHARLOOKS handle, this will be used in place of any candidate looks data structure. Usually, you will use this function when you must control the appearance of text that is typed in by the user.

1.2.3.16 The Default Character Looks

You may specify the default character looks (e.g., font, size, family, etc.) that are to be used when entering or displaying text if the text itself does not have explicit character looks. You do so by assigning a value to the LOOKS property which may be:

- a font descriptor
- a property list that is acceptable to TEDIT.LOOKS
- a CHARLOOKS handle

1.3 TEdit Data Structures

TEdit uses a large number of data structures in order to manipulate the text displayed in its editing window. All of these data structures are described in this section with examples of the values that their fields might take.

1.3.1 The Text Stream

The *Text Stream* is a record structure that describes the source of text that is displayed in the TEdit Window. Typically, this source will be a file stored on an external disk. The fields of the record structure are described in Table 1-4.

Table 1-4. Text Stream Record Structure

Field Name	Usage
REALFILE	The file which underlies the current piece of text.
CHARSLEFT	The number of characters that will be left in the current piece of text the next time the file crosses a page boundary.
TEXTOBJ	The handle of the Text Object that is editing this text.
PIECE	The handle of the piece from which characters are currently being fetched or to which they are being put.
PCNO	The position of the piece in the piece table.
PCSTARTPG	The underlying file page number that this piece starts on.
PCSTARTCH	The character within the page of the underlying file that this piece starts on.
PCOFFSET	The offset into the current piece, as of the last page crossing.
CURRENTLOOKS	The CHARLOOKS object that is currently applicable to the characters being taken from the stream.
CURRENTPARALOOKS	The FMTSPEC object that is currently applicable to the characters being taken from the stream.

TEdit

CURRENTIMAGESTREAM	The image stream object to which the text is currently being sent for display.
LOOKSUPDATEFN	A function that is called each time you change the character looks of a piece of text.
FATSTREAMP	T, if the current piece is 16 bit characters.

A Text Stream is created by the function `OPENTEXTSTREAM`. The usual stream operations may be applied to this data structure: `BIN`, `SETFILEPTR`, `GETFILEPTR`, and `GETEOFPTR`. `BOUT` inserts a character into the stream at the current position of the file pointer. When editing in a window, the stream handle can be determined using the function `TEXTSTREAM`.

1.3.2 The Text Object

A *Text Object* is a record structure used by TEdit to store its state information about the text that is currently being edited. The fields of the record structure are described in Table 1-5..

Table 1-5. Text Object Record Structure

Field	Description
\DIRTY	A flag indicating whether the object has been modified or not.
PCTB	The piece table (pointer to the first entry)
TEXTLEN	The current number of characters in the text.
\INSERTPC	A piece object that holds characters typed by the user.
\INSERTPCNO	The piece number of the input piece.

TEdit

\INSERTNEXTCHAR	The character number of the next character which is typed into that piece.
\INSERTLEFT	The amount of space left in the type-in piece.
\INSERTLEN	The number of characters already in the piece.
\INSERTSTRING	The string which the piece describes.
\INSERTFIRSTCH	The character number in the text of the first character in the piece.
\INSERTPCVALID	T, if its okay to use the cached piece, but set to NIL if you require that each insertion or deletion require a new piece

Table 1-5. Text Object Record Structure (Continued)

Field	Description
\WINDOW	The window(s) where the text object is displayed. If its value is NIL, then there is no edit window for this text. There will be more than one window if the main editing window has been divided into several panes.
MOUSEREGION	The section of the edit pane that the mouse is in.
LINES	A pointer to the top of a chain of line descriptors for the displayed text.
DS	The display stream where the text object is displayed.
SEL	The current selection within the text object; actually, the handle of its record structure.
SCRATCHSEL	The scratch space for the selection code; actually, a pointer to a temporary SELECTION object.
MOVESEL	The source of the next MOVE of text.
SHIFTEDSEL	The source for the next COPY.
DELETESEL	The text to be deleted very soon now.
WRIGHT	The right edge of the window where this text is displayed.
WTOP	The top of the window or region.
WBOTTOM	The bottom of the window or region.
WLEFT	The left edge of the window or region.

TEdit

TXTFILE	The original text file that is currently being edited.
\XDIRTY	T if the text object was changed since it was last saved.
STREAMHINT	A pointer to the TEXTOFD stream which gives access to this text object.
EDITFINISHFLG	T, if the user has requested the closing of the editor. TEdit halts after the next pass through the polling loop if this flag is set. No check is made for unsaved changes. Unless it is T, the value of EDITFINISHEDFLG will be returned as the result of TEdit (unless it was called with DON'TSPAWN equal to T).
CARET	Describes the flashing caret for the editing window.
CARETLOOKS	The font to be used for inserted text.
WINDOWTITLE	The title of the TEdit editing window.
THISLINE	A data structure handle (see below) describing the current line in the object.
MENUFLG	T, if this text object is a TEdit-style menu
FMTSPEC	The default formatting specification to be used when formatting paragraphs.
FORMATTEDP	T, if this document will contain paragraph formatting information.

TEdit

Table 1-5. Text Object Record Structure (Continued)

Field	Description
TXTREADONLY	Indicates text is subject to shift selection only.
TXTTERMSA	Special instructions for displaying characters on the screen.
EDITOPACTIVE	T, if there is an editing operation in progress; it is used to interlock the TEdit menu.
DEFAULTCHARLOOKS	The default character looks, if any, to be applied to characters entering the file from outside (i.e., typing).
TXTRTBL	The read table to be used by the command loop for command dispatch.
TXTWTBL	The read table to be used to decide when a word breaks in text.
EDITPROPS	The PROPSO that were passed to TEdit.
BLUEPENDINGDELETE	T, if the next insertion into a document is to be preceded by the deletion of the then-current selection.
TXTHISTORY	The history list for this session of TEdit.
SELWINDOW	The window in which the last real selection was made for the edit. It is used to control the placement of the caret.

TEdit

PROMPTWINDOW	A window used for unscheduled interactions with the user.
DISPLAYCACHE	The bitmap that is used when building an image of a line to be displayed.
DISPLAYCACHEDS	The display stream that is used to build line images.
DISPLAYHCPYDS	The display stream that is used to build images for hardcopy simulation mode.
TXTPAGEFRAMES	A tree of page frames which specified how the document is to be laid out.
TXTNEEDSUPDATE	T, if the screen contains an invalid copy of the text.
TXTCCHARLOOKSLIST	A list of all the CHARLOOKS in the document so that they may be kept unique.
TXTPARALOOKSLIST	A list of all the FMTSPECs in the document so that they can be kept unique.

Consider the following example of a text object for the text being edited by TEdit in Section 1.2.

\DIRTY	T
PCTB	{ARRAYP}#71,175260
TEXTLEN	494
\INSERTPC	{PIECE}#54,72642
\INSERTPCNO	4
\INSERTNEXTCH	23
\INSERTLEFT	476
\INSERTLEN	0

TEdit

\INSERTSTRING	""
\INSERTFIRSTCH	23
\INSERTPCVALID	NIL
\WINDOW	({WINDOW}#74,25000)
MOUSEREGION	TEXT
LINES	({LINEDESCRIPTOR}#54,73542)
DS	NIL
SEL	{SELECTION}#54,114462
SCRATCHSEL	{SELECTION}#54,114524
MOVESEL	{SELECTION}#54,114566
SHIFTEDSEL	{SELECTION}#54,114630
DELETSEL	{SELECTION}#54,114672
WRIGHT	530
WTOP	237
WBOTTOM	0
WLEFT	0
TXTFILE	"A complex number is a generalization of ..."
\XDIRTY	T
STREAMHINT	{STREAM}#60,46150
EDITFINISHEDFLG	NIL
CARET	{TEDITCARET}#55,136566
CARETLOOKS	{CHARLOOKS}#56,63020
WINDOWTITLE	NIL
THISLINE	{THISLINE}#56,117720
MENUFLG	NIL
FMTSPEC	{FMTSPEC}#56,61524
FORMATTEDP	NIL
TXTREADONLY	NIL
TXTERMSA	NIL
EDITOPACTIVE	NIL
DEFAULTCHARLOOKS	{CHARLOOKS}#56,63020
TXTRTBL	NIL
TXWTBL	NIL
EDITPROPS	(CACHE NIL)

TEdit

```
BLUEPENDINGDELETE  NIL
TXTHISTORY          (Insert LEFT 0 23
                    {PIECE}#54,72642 NIL NIL NIL)
SELWINDOW           {WINDOW}#74,25000
PROMPTWINDOW        {WINDOW}#57,141320
DISPLAYCACHE        {LINCACHE}#56,106770
DISPLAYCACHEDS      {STREAM}#57,55234
TXTPAGEFRAMES       NIL
TXTNEEDSUPDATE      NIL
TXTCHARLOOKSLIST    ({CHARLOOKS}#56,63020)
TXTPARALOOKSLIST    ({FMTSPEC}#56,61524)
```

Figure 1-14. TEdit Text Object Example

1.3.3 The Selection Object

The *Selection Object* is a record structure that maintains information about the current selection in the text. It is described in Table 1-6.

Table 1-6. The Selection Object

Field Name	Usage
Y0	The Y-coordinate of the topmost line of the selection
X0	The X-coordinate of the leftmost edge of the topmost line of the selection.
DX	The width of the selection, if it is on one line.
CH#	The character number of the first selected character. The first character in the text is numbered beginning with one.
XLIM	The X-coordinate of the right edge of the last character in the selection. If DCH is equal to 0, then XLIM is equal to X0.

TEdit

CHLIM	The character number of the first character which occurs just after the last selected character; it must be as large as CH#.
DCH	The number of characters selected; it may be zero to indicate a point selection whence it points to where characters will be inserted into the text.
L1	A pointer to the line descriptor for the line where the first selected character resides.
LN	A pointer to the line descriptor for the line which contains the last character of the selection.
YLIM	The Y-coordinate of the bottom of the line that ends the selection.
POINT	The location where the caret should appear; it takes the values LEFT or RIGHT.
SET	T, if this selection is real; otherwise, NIL.
\TEXTOBJ	A pointer to the text object that describes the selected text.
SELKIND	The type of selection; it takes the values CHAR, PARA, WORD, LINE.
HOW	The texture used to highlight the selection.
HOWHEIGHT	The height of the highlight; usually 1.
HASCARET	T, if there should be a caret for this selection.
SELOBJ	A pointer to another selection object if this selection is contained within another one.
ONFLG	T, if the selection is to be highlighted on the screen.
SELOBJINFO	A storage area for information about other selections within this selections.

The selection object for the word "generalization" in the text has the following structure:

Y0	225
X0	162

TEdit

DX	98
CH#	23
XLIM	260
CHLIM	37
DCH	14
L1	({LINEDESCRIPTOR}#54,73524)
LN	({LINEDESCRIPTOR}#54,73524)
YLIM	225
POINT	RIGHT
SET	T
\TEXTOBJ	{TEXTOBJ}#54,115550
SELKIND	WORD
HOW	65535
HOWHEIGHT	1
HASCARET	T
SELOBJ	NIL
ONFLG	NIL
SELOBJINFO	NIL

Figure 1-15. TEdit Selection Object Example

1.3.4 The Line Descriptor Object

A *Line Descriptor Object* is used to describe a line of text within the TEdit editing window. Its structure is described in Table 1-7.

Table 1-7. The Line Descriptor Object

Field Name	Usage
YBOT	The Y value for the bottom of the line.
YBASE	The Y value for the base line on which the characters sit.
LEFTMARGIN	The left margin of the line (in pixels).
RIGHTMARGIN	The right margin of the line (in pixels).

TEdit

LXLIM	The X value of the right edge of the rightmost character on the line.
SPACELEFT	The space left on the line, ignoring trailing blanks and <CR>s.
LHEIGHT	The total height of the line.
ASCENT	The ascent of the line above YBASE.
DESCENT	The descent of the line below YBASE.
LTRUEDESCENT	The true descent for this line, unadjusted for the line leading.
LTRUEASCENT	The true ascent for this line, unadjusted for pre-paragraph leading.
CHAR1	The character index of the first character on the line.
CHARLIM	The character index of the last character on the line.
CHARTOP	The character index of the character which forced the line break.
NEXTLINE	A pointer to the next line descriptor.
PREVLINE	A pointer to the previous line descriptor.
LMARK	The type of special line marker to be displayed in the left margin for this paragraph; should be one of SOLID, GREY, or NIL.
LTEXTOBJ	A cached text object from which this line took its text; it is used in generating hardcopy when disambiguation is required.
CACHE	A cached THISLINE which stores information for hardcopy processing.
LDOBJ	The object which lies behind this line of text; it is used for updating, etc.
LFMTSPEC	The format specification for this line's paragraph.
DIRTY	A flag which is T if this line was changed since it was last formatted.
CR\END	A flag which is T if this line ends with a CR.

TEdit

DELETED	A flag which is T if this line has been completely deleted since it was last formatted or displayed.
LHASPROT	Indicates this line contains protected text.
LHASTABS	The relative character index for the final tab in the line, if any.
1STLN	A flag indicating if this is the first line of a paragraph.
LSTLN	A flag indicatng if this is the last line of the paragraph.

The values for the line descriptor object containing the selection "generalization" are described in the following figure.

YBOT	225
YBASE	228
LEFTMARGIN	8
RIGHTMARGIN	522
LXLIM	519
SPACELEFT	10
LHEIGHT	12
ASCENT	9
DESCENT	3
LTRUEDESCENT	3
LTRUEASCENT	9
CHAR1	1
CHARLIM	73
CHARTOP	74
NEXTLINE	{LINEDESCRIPTOR}#54,73576
PREVLINE	{LINEDESCRIPTOR}#54,73452
LMARK	NIL
LTEXTOBJ	NIL
CACHE	NIL
LDOBJ	NIL
LFMTSPEC	{FMTSPEC}#56,61524
DIRTY	NIL

TEdit

CR\END	NIL
DELETED	NIL
LHASPROT	NIL
LHASTABS	NIL
1STLN	T
LSTLN	NIL

Figure 1-16. TEdit Line Descriptor Object Example

1.3.5 The Piece Object

A *piece* describes a string, part of a file, or a generalized object. A PIECE object is used to describe the piece of text at the current file pointer. Its structure is described in Table 1-8.

Table 1-8. The Piece Object

Field Name	Usage
PSTR	The string where this piece's text resides; NIL indicates
PFILE	The file which contains this piece's text; NIL indicates
PFPOS	The file pointer to the start of the piece in the file; 0 if PFILE is NIL.
PLEN	The length of the piece, in characters.
NEXTPIECE	The next piece of this text object.
PREVPIECE	The prior piece of this text object.
PLOOKS	The formatting information for this piece.
POBJ	The object handle of the object this piece describes.
PPARALAST	A flag indicating this piece contains a paragraph
PPARALOOKS	The paragraph looks for this piece.
PNEW	A flag indicating this piece is a new piece of text.

TEdit

PFATP	T, if the characters are fat (e.g., 16 bits each).
-------	--

An example of a piece object depicted in the following figure.

```
PSTR          ""
PFILE         NIL
PFPOS         0
PLEN          0
NEXTPIECE     {PIECE}#54,72620
PREVPIECE     {PIECE}#54,140066
PLOOKS        {CHARLOOKS}#56,63020
POBJ          NIL
PPARALAST     NIL
PPARALOOKS    {FMTSPEC}#56,61524
PNEW          T
PFATP         NIL
```

Figure 1-17. TEdit Piece Object Example

1.3.6 The THISLINE Object

A *THISLINE object* describes a particular line within a piece of text. It takes the following structure.

Table 1-9. Structure of the THISLINE Object

Field Name	Usage
DESC	A line descriptor for the line this object describes now.
LEN	The length of the line in characters.
CHARS	An array of character codes (or objects) in the line; a character code of 400 indicates a dummy entry which is used for a character looks change; the next entry is stored in LOOKS.

TEdit

WIDTHS	An array of the character's widths in points which us
LOOKS	An array of any character looks within the line; LOOKS(0)

An example of a THISLINE object is depicted in the following figure.

```
DESC          {LINEDESCRIPTOR}#54,73524
LEN           72
CHARS        {ARRAYP}#55,27120
WIDTHS       {ARRAYP}#55,27110
LOOKS        {ARRAYP}#55,27114
```

Figure 1-18. TEdit THISLINE Object Example

1.3.7 The Line Cache Object

The *Line Cache object* is used to store bitmaps of the characters in a line of text. It has the following structure.

Table 1-10. Structure of the Line Cache Object

Field Name	Usage
LCBITMAP	The bitmap that will be used by this instance of the
LCNEXTCACHE	A pointer to the next cache object; used to simplify

An example of a Line Cache object is depicted in the following figure.

```
LCBITMAP     {BITMAP}#61,166446
LCNEXTCACHE  {LINECACHE}#56,106770
```

Figure 1-19. Line Cache Object Example

1.3.8 The Character Looks Object

Character looks pertain to how individual characters are formatted relative to the rest of the text. TEdit supports the following character looks:

- type face
- type style
- type size
- positioning of individual characters

Character looks for a selection are stored in a CHARLOOKS object which has the structure depicted in Table 1-19.

Table 1-11. The Character Looks Object

Field Name	Usage
CLFONT	The font descriptor for the characters.
CLNAME	The name of the font (e.g., Gacha).
CLSIZE	The font size in points.
CLBOLD	A flag indicating if the characters are displayed in bold
CLULINE	A flag indicating if the characters are to be
CLOLINE	A flag indicating if the characters are to be
CLSTRIKE	A flag indicating if the characters are to be struck
CLOFFSET	An offset which is used to specify superscripting
CLSMALLCAP	A flag indicating small capitals.
CLINVERTED	A flag indicating if the characters are to be
CLPROTECTED	A flag indicating the characters can't be selected
CLINVISIBLE	A flag indicating that TEdit is to ignore these

TEdit

CLSELHERE	T, if TEdit can put a selection after this character.
CLCANCOPY	T, if this text can be selected for copying even
CLSTYLE	The style to be used in marking these characters.
CLUSERINFO	A user data storage area.
CLLEADER	A storage area for creating a leader string for the
CLRULES	A list of rules for horizontal positioning.
CLMARK	A flag used for marking and sweeping when the text is put to a file. T indicates these character looks are really in use in the text.

1.3.9 The Format Specification Object

The *Format Specification object* is used to describe the paragraph looks of a segment of text in a document. It has the following structure:

Table 1-12. Structure of the Format Specification

Field Name	Usage
1STLEFTMAR	The left margin of the first line of the paragraph.
LEFTMAR	The left margin of the rest of the lines of the
RIGHTMAR	The right margin of the paragraph.
LEADBEFORE	The leading space above the paragraph's first line
LEADAFTER	The leading space below the paragraph's bottom line (in points); however, this is not currently used in TEdit.
LINELEAD	The leading space between lines (in points); TEdit actually adds this value

TEdit

	below each line in the paragraph including the last line.
FMTBASETOBASE	The baseline-to-baseline spacing between lines in this paragraph; this value overrides the line leading, if specified.
TABSPEC	A list of tab stops for this paragraph; the CAR specifies the default tab width.
QUAD	How the paragraph is to be formatted; should be one of LEFT, RIGHT, CENTERED, or JUSTIFIED.
FMTSTYLE	The STYLE that controls the paragraph's appearance.
FMTCHARSTYLES	The character styles that control the appearance of characters in this paragraph.
FMTUSERINFO	Space for user-defined properties and information.
FMTSPECIALX	A special horizontal location on the printed page
FMTSPECIALY	A special vertical location on the printed page for
FMTHEADINGKEEP	Indicates that this paragraph should be kept within the top line of the next paragraph.
FMTPARATYPE	The type of paragraph: TEXT, PAGEHEADING, etc.
FMTPARASUBTYPE	The subtype of this paragraph, e.g., what kind of page heading it is.
FMTNEWPAGEBEFORE	Indicates that a new page should be started before this paragraph.
FMTNEWPAGEAFTER	Indicates that a new page should be started after this paragraph.
FMTKEEP	Information about how this paragraph is to be kept with other paragraphs.
FMTCOLUMN	The number of columns if side-by-side paragraphs is specified.

TEdit

FMTVERTRULES	The vertical rules in force.
FMTMARK	Records the paragraph looks that are actually used and, therefore, should be written to a file when a Put command is executed.
FMTHARDCOPY	T, if this paragraph is to be displayed in hardcopy format; NIL means this paragraph is invisible in hardcopy.

1.4 TEdit User Interface Functions

TEdit provides functions at the user interface which allow you to execute any of the TEdit commands on a text stream which you specify. This mode allows you to use TEdit as a programmable editor within a program.

1.4.1 Opening a Text Stream

A text stream may be opened on a piece of text using the function **OPENTEXTSTREAM**, which takes the form:

Function:	OPENTEXTSTREAM
# Arguments:	5
Arguments:	1) TEXT, a text object descriptor 2) WINDOW, a window handle 3) START, the start of the text 4) END, the end of the text 5) PROPS, a list of TEdit properties
Value:	A text stream handle.

TEdit

`OPENTEXTSTREAM` creates a text stream object describing `TEXT` and returns it to the caller.

If `WINDOW` is specified, the text will be displayed in the window. Moreover, any changes made to the text will be displayed in the window as they occur. Thus, even though editing of text may occur under the control of some function, you may still observe the edits as they occur.

`TEXT` may be an existing text object (e.g., a `TEXTOBJ`) or text stream. `PROPS` has the same values as specified for `TEDIT`. Consider the following example:

```
<-(SETQ MYTEXTSTREAM
  (OPENTEXTSTREAM MYTEXT AWINDOW))
{STREAM}#64,107554
```

If `START` and `END` are given, they specified the portion of text to be edited. If the text object is a file having `TEDIT` characteristics, these will be reproduced in the window. Otherwise, the object is treated as a plain-text object. Typically, `START` and `END` are specified for files where you only want to edit a portion of the file.

1.4.2 Creating a Text Stream

You may determine the text stream associated with some `TEDIT` object using the function `TEXTSTREAM`, which takes the form:

Function:	<code>TEXTSTREAM</code>
# Arguments:	1
Arguments:	1) <code>TEXTOBJ</code> / <code>WINDOW</code> , a text object or window handle
Value:	A stream handle.

TEdit

TEXTOBJ/WINDOW can be a text object, a text stream, a process in which TEdit is running, or a TEdit editing window. If it is any of these, TEXSTREAM returns the associated text stream handle. Consider the following example:

```
<-(SETQ mytextstream (TEXTSTREAM awindow))  
{STREAM}#60,47150
```

The argument may not be a string.

1.4.2 Making a Selection in a Text Stream

Given a text stream, you may make a selection in the text stream using the function **TEDIT.SETSEL**, which takes the form:

Function:	TEDIT.SETSEL
# Arguments:	7
Arguments:	1) STREAM, a text stream handle 2) CH#orSEL, a selection handle or character index 3) LEN, the number of characters to select 4) POINT, the side of the selection on which the caret is placed 5) PENDINGDEL?, a flag specifying a delete is pending 6) LEAVECARETLOOKS, a flag specifying how inserted characters will look
Value:	A stream handle.

TEdit

TEDIT.SETSEL sets the selection in STREAM. If CH#orSEL is a selection object, it is used directly. Otherwise, CH#orSEL represents the first character in the selection. LEN is then interpreted as the number of characters to select from that first character. If LEN is zero, then CH#orSEL just specifies an insertion point where text will be inserted. Consider the following examples:

```
<-(TEDIT.SETSEL MYTEXTSTREAM 11 6 'RIGHT T)
{STREAM}#56,127734
```

Note: You should remember that character pointers start with the integer 1 while file indexing starts with the number 0.

POINT specifies which side of the selection that the caret should be displayed in the window. It must be one of the atoms LEFT or RIGHT.

If PENDINGDEL? is non-NIL, the selected text is marked for pending deletion. That is, it will be deleted and overwritten by the next type-in (or if the text is moved or copied).

When you make a selection and then insert new text, the inserted text has the same characteristics as the text just selected. You may suppress this feature by setting the flag LEAVECARETLOOKS to a non-NIL value.

Selections may be made for different purposes. OPERATION allows you to specify the purpose of a selection. It may be one of the following atoms:

- NORMAL
- MOVE
- COPY
- PENDINGDEL
- DELETE

TEdit

- INVERTED

The latter operations just highlights the selected text and leaves the caret flashing.

1.4.3 Getting the Current Selection

Once a selection has been made in a text stream, you may retrieve it using the function **TEDIT.GETSEL**, which takes the form:

Function:	TEDIT.GETSEL
# Arguments:	1
Arguments:	1) STREAM, a text stream handle
Value:	A selection handle.

TEDIT.GETSEL returns a copy of the current selection in the edit window which is described by STREAM. Consider the following example:

```
<-(SETQ myselection (TEDIT.GETSEL mytextstream))  
{SELECTION}#54,114420
```

In Figure 1-13, the word "generalization" is underlined. The SELECTION object returned by TEDIT.GETSEL corresponds to that word.

1.4.4 Showing a Selection

If the current selection is displayed in a window, you can highlight the selection using the function **TEDIT.SHOWSEL**, which takes the form:

Function:	TEDIT.SHOWSEL
# Arguments:	3

TEdit

Arguments: 1) STREAM, a text stream handle
 2) ONFLG, a flag specifying highlighting
 3) SELECTION, a selection handle
Value: NIL

If ONFLG is T (or some non-NIL value), the selection SELECTION in the text stream specified by STREAM will be highlighted. If NIL, any highlighting is turned off. Consider the following example:

```
<-(TEDIT.SHOWSEL mystream T myselection)  
NIL
```

The result is depicted in Figure 1.14.

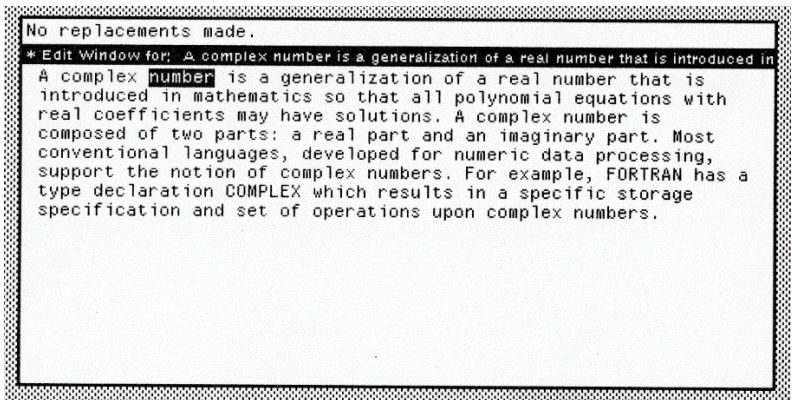


Figure 1-20 TEDIT.SHOWSEL Example

If SELECTION is NIL, then the function is applied to the current selection of STREAM.

TEdit

1.4.5 Inserting into a Text Stream

A text string may be inserted into a text stream using the function **TEDIT.INSERT**, which takes the form:

Function:	TEDIT.INSERT
# Arguments:	5
Arguments:	1) STREAM, a text stream handle 2) TEXT, a text string 3) CH#orSEL, a selection object handle 4) LOOKS, the font for the text 5) DONTSCROLL, a flag controlling window scrolling
Value:	NIL.

TEDIT.INSERT inserts the string TEXT into the text stream STREAM as though you had typed it in. CH#orSEL specifies the character number where the text is to be inserted as shown in Table 1-13.

Table 1-13. TEDIT.INSERT options

Field Name	Usage
NIL	The text is inserted at the current location of the caret.
a number	The text is inserted before the character in the text stream.
a SELECTION	The text is inserted according to the selection

Consider the following example whose result is depicted in Figure 1-21.

```
<-(TEDIT.INSERT MYTEXTSTREAM "in the normal  
definition " 18 NIL)
```

NIL

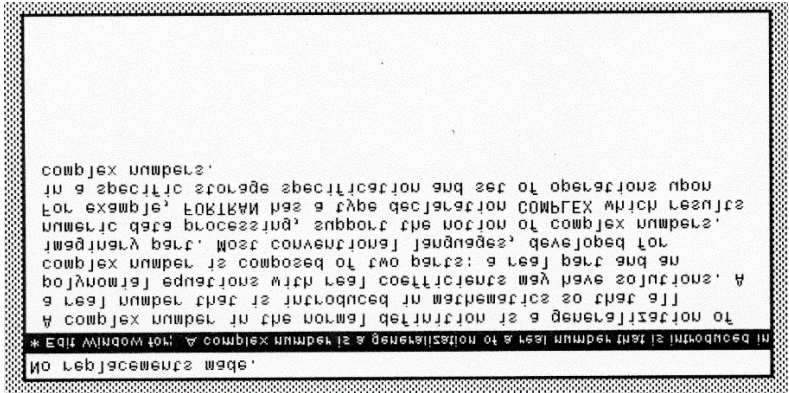


Figure 1-21. TEDIT.INSERT Example

If the location to insert is NIL, TEdit inserts the string before the caret. Consider the following example which inserts "mathematical" before "generalization" because that is the current location of the caret (see Figure 1-22).

```
<-(TEDIT.INSERT mystream "mathematical" NIL NIL)
NIL
```

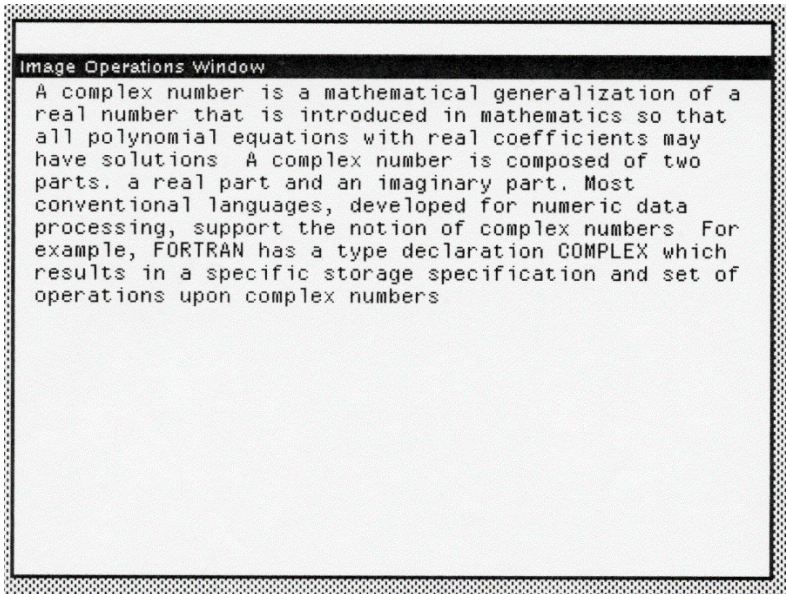



Figure 1-22. TEDIT.INSERT with NIL selection

If LOOKS is specified, it must be a font descriptor which specifies how the text will be displayed. This allows you to dynamically change the looks for a small piece of text upon insertion.

Normally, TEdit scrolls the edit window so that all changes to the text are made visible as they occur. You may suppress scrolling by setting DONTSCROLL to a non-NIL value.

1.4.6 Deleting Text from a Stream

Text may be deleted from a text stream using the function **TEDIT.DELETE**, which takes the form:

TEdit

Function:	TEDIT.DELETE
# Arguments:	3
Arguments:	1) STREAM, a stream handle 2) CH#orSEL, a selection handle 3) LEN, the number of characters to delete
Value:	NIL.

TEDIT.DELETE deletes the specified text selection from the text stream. CH#orSEL specifies the selection to be deleted. It may be a number indicating the character at which the selection is to begin or a text selection object. If it is a number, then LEN must be present to specify the number of characters to delete. Consider the following example:

```
<-(TEDIT.DELETE MYTEXTSTREAM 18 25)  
NIL
```

This command deleted the text "in the normal definition, which was inserted by the example for TEDIT.INSERT above.

1.4.7 Finding Text in a Text Stream

You can search for the next occurrence of a piece of text inside a text stream using the function **TEDIT.FIND**, which takes the form:

Function:	TEDIT.FIND
# Arguments:	5
Arguments:	1) STREAM, a stream handle 2) TEXT, a text string 3) START#, a character index 4) END#, a character index 5) WILDCARDS?, a flag
Value:	The index of the first character of

TEdit

TEXT, if found.

TEDIT.FIND searches for the next occurrence of TEXT in STREAM. START# and END# specify the boundaries of the search area. If START# is NIL, the search will start at the beginning of the text stream. If END# is NIL, the search continues to the end of the text stream.

TEDIT.FIND returns the character index of the first character of TEXT which matches a sequence of characters in the text stream. If no match is found, TEDIT.FIND will return NIL. Consider the following examples:

```
<-(TEDIT.FIND MYTEXTSTREAM "numeric")  
266
```

```
<-(TEDIT.FIND MYTEXTSTREAM "FORTRAN" 50 200)  
NIL
```

In the first example the search was unbounded, so TEdit found the string "numeric" beginning at location 266 of the text. In the second example, I specified that TEdit should look only between character positions 50 and 200. Because the string "FORTRAN" begins at location 363, it was not found by this search request.

WILDCARDS?, if non-NIL, specifies the wild card characters:

- # Matches any single character
- * Matches any sequence of characters
- ' Is used to quote one of the wild cards.

When wild cards are used, TEdit returns a list consisting of the first and last characters of the matching sequence in the text stream. Consider the following example:

TEdit

```
<-(TEDIT.FIND MYTEXTSTREAM "co#plex" NIL NIL '#)  
(167 173)
```

Here, TEdit returns a list of the beginning and ending character indices of the string which matched the TEXT specification.

1.4.8 Generating Hardcopy of Text

You can obtain a hardcopy of a piece of text using the function **TEDIT.HARDCOPY**, which takes the form:

Function:	TEDIT.HARDCOPY
# Arguments:	6
Arguments:	1) STREAM, a stream handle 2) FILE, a file name 3) DONTSEND, a printer flag 4) BREAKPAGETITLE, the title for the break page 5) SERVER, a print server name 6) PRINTOPTIONS, a list of options
Value:	NIL.

TEDIT.HARDCOPY sends the text stream to the printer specified by SERVER. If SERVER is NIL, then TEdit uses the printer specified by DEFAULTPRINTINGHOST.

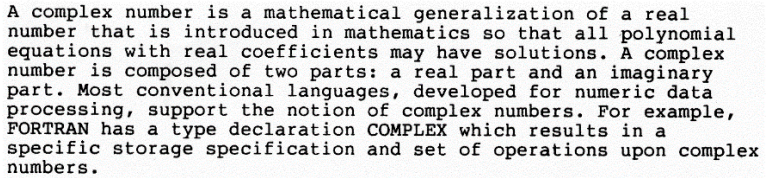
PRINTOPTIONS allows you to specify printing options specific to the printer. If FILE is given, the Press file will be left available for you to use later.

DONTSEND indicates that the file should not be sent to the printer, but merely created for later use. Consider the following examples:

TEdit

```
<-(TEDIT.HARDCOPY mystream)
NIL
```

The result is depicted in Figure 1-23.



A complex number is a mathematical generalization of a real number that is introduced in mathematics so that all polynomial equations with real coefficients may have solutions. A complex number is composed of two parts: a real part and an imaginary part. Most conventional languages, developed for numeric data processing, support the notion of complex numbers. For example, FORTRAN has a type declaration `COMPLEX` which results in a specific storage specification and set of operations upon complex numbers.

Figure 1-23. TEDIT.HARDCOPY Example

When TEdit has finished formatting and transmitting all pages, it notifies you with a message in the prompt pane.

If `BREAKPAGETITLE` is non-NIL, it is used as the title on the break page printer before the text. Unfortunately, my 4045 laser printer did not support this feature.

1.4.9 Changing the Looks of Selected Characters

You can change the looks of selected characters using the function `TEDIT.LOOKS`, which takes the form:

Function	TEDIT.LOOKS
# Arguments:	4
Arguments:	1) STREAM, a stream handle 2) NEWLOOKS, a list in property format of the looks specifications 3) SELORCH#, a selection handle 4) LEN, the number of characters

TEdit

Value: NIL.

TEDIT.LOOKS changes the character looks of the text selection in the text stream. SELORCH# may be:

- a SELECTION object which specifies the text to be changed
- an integer must be a FIXP which specifies the first character to be changed;
- LEN must also be present
- NIL indicates the current selection is to be used

NEWLOOKS is a list of property-value pairs which describe how the text selection should look. Any properties not explicitly mentioned in NEWLOOKS retain their old values. The properties which may appear in NEWLOOKS are presented in Table 1-14.

Table 1-14. NEWLOOKS Property Values

Property Value	Description
FAMILY	A name of a font family in which the selected text will appear.
FACE	The face of the new font; the value is like the one accepted by FONTCREATE.
WEIGHT	The new eight of the font - either LIGHT, MEDIUM, or BOLD.
SLOPE	The new slope of the font - either REGULAR or ITALIC; it overrides the SLOPE parameter of the font descriptor.
EXPANSION	The new weight for the font - either CONDENSED, REGULAR, or EXPANDED.
SIZE	The new point size.
UNDERLINE	Either ON or OFF, but specifies whether or not the text is underlined.

TEdit

OVERLINE	Either ON or OFF, but specifies whether or not the text will be overscored.
STRIKEOUT	Either ON or OFF, but specifies whether or not the text will be displayed with a line struck through it.
SUPERSCRIPT	A distance, in points, by which text will be raised above the normal baseline.
SUBSCRIPT	A distance, in points, by which the text will be lowered below the normal baseline.
PROTECTED	Either ON or OFF, but specifies whether the text can be selected by the mouse or not.
SELECTPOINT	Either ON or OFF, but specifies whether or not the user can make point selection after it, even if the text is protected.
INVISIBLE	Either ON or OFF, but specifies whether or not the text appears on the screen or in hardcopy.
INVERTED	Either ON or OFF, but indicates whether or not the text appears in reverse video or not.

Consider the following examples:

```
<-(TEDIT.LOOKS mystream '(FAMILY HELVETICA  
WEIGHT BOLD) 23 27)  
NIL
```

which modifies the looks of the string "mathematical generalization" as depicted in Figure 1-24..

TEdit

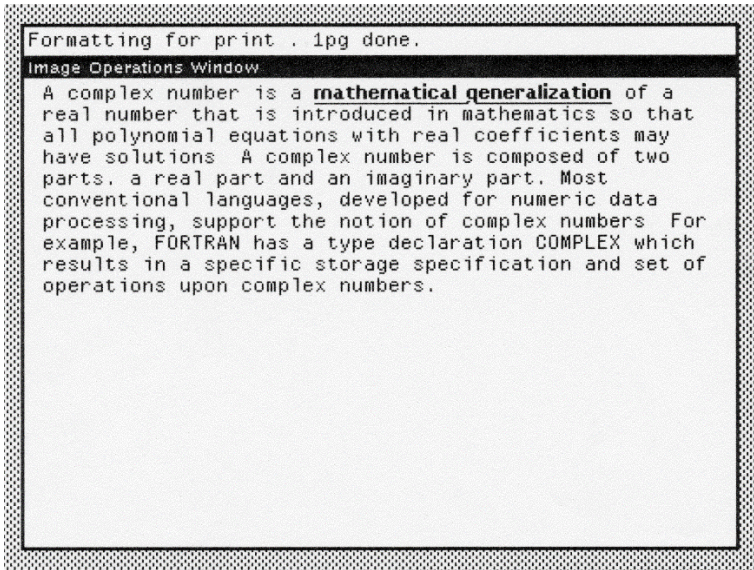


Figure 1-24. Example 1 of TEDIT.LOOKS

```
<-(TEDIT.LOOKS mystream
      '( UNDERLINE ON
        INVERTED ON
        FACE ITALIC)
      56 11)
```

NIL

which underlines the string "real number" as depicted in Figure 1-25.

TEDIT

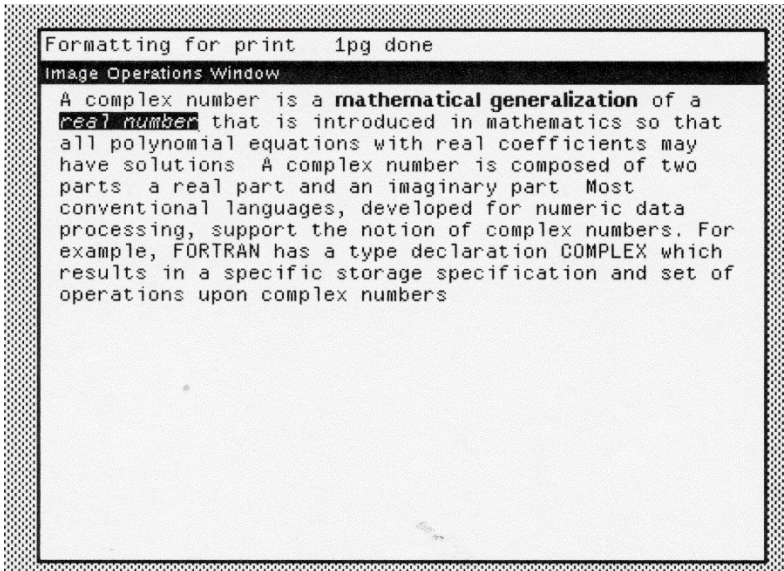


Figure 1-25. Example 2 of TEDIT.LOOKS

1.4.10 Getting the Character Looks of a Selection

You may obtain a list of the current set of character looks for a selection using the function **TEDIT.GET.LOOKS**, which takes the form:

Function:	TEDIT.GET.LOOKS
# Arguments:	2
Arguments:	1) STREAM, a text stream handle 2) SELORCH#, a selection specification
Value:	A list of character looks.

TEDIT.GET.LOOKS returns a list of character looks (in property list format) which may subsequently be passed to

TEdit

TEDIT.LOOKS. SELORCH# may be a selection object, an integer, or NIL - whence the current selection is used. Consider the following examples:

```
<-(TEDIT.GET.LOOKS mytextstream myselection)
(SUPERSCRIPT 0
INVISIBLE OFF
SELECTPOINT OFF
PROTECTED OFF
SIZE 10
FAMILY GACHA
OVERLINE OFF
STRIKEOUT OFF
UNDERLINE OFF
EXPANSION REGULAR
SLOPE REGULAR
WEIGHT MEDIUM
INVERTED OFF
USERINFO NIL
STYLE NIL)
```

1.4.11 Copying Character Looks

You may change the character looks of one piece of text based on those of another piece of text by copying the character look specifications. To do so, you use the function **TEDIT.COPY.LOOKS**, which takes the form:

Function:	TEDIT.COPY.LOOKS
# Arguments:	3
Arguments:	1) STREAM, a text stream handle 2) SOURCE, a selection specification 3) DEST, a selection specification
Value:	NIL.

TEdit

TEDIT.COPY.LOOKS makes the character looks of DEST appear the same as those of SOURCE. SOURCE and DEST may be text selection objects or integers. If DEST is a selection object, it must be in STREAM, whereas SOURCE can be in any text stream. Consider the following examples where MYSEL1 is "real number" and MYSEL2 is "mathematical generalization":

```
<-(SETQ MYSEL1 (TEDIT.GETSEL mystream))  
{SELECTION}#55,122734
```

```
<-(SETQ MYSEL2 (TEDIT.GETSEL mystream))  
{SELECTION}#55,122672
```

```
<-(TEDIT.COPY.LOOKS mystream mysel1 mysel2)  
NIL
```

The result is depicted in Figure 1-26.

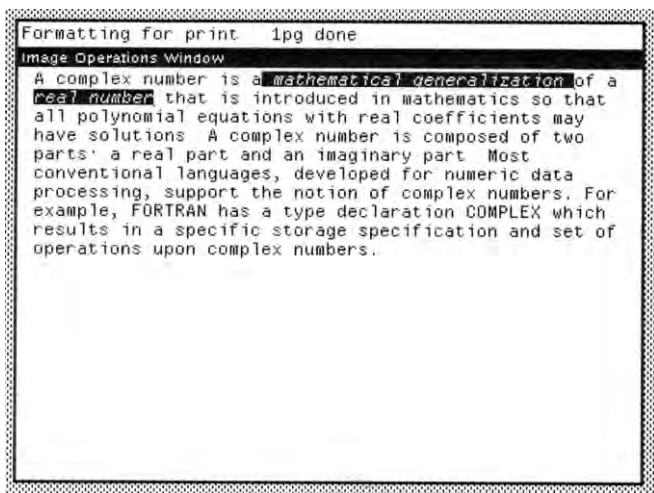


Figure 1-26. Example of TEDIT.COPY.LOOKS

TEdit

1.4.12 Quitting TEdit

You may exit TEdit by executing the function **TEDIT.QUIT**, which takes the following form:

Function:	TEDIT.QUIT
# Arguments:	1
Arguments:	1) STREAM, a stream handle
Value:	NIL.

TEDIT.QUIT terminates the TEdit process (if any) after writing the contents of the Edit Pane to the appropriate data structure. It also closes the TEdit Window.

```
<-(TEDIT.QUIT MYSTREAM)  
NIL
```

1.5 TEdit System Variables

TEdit operation is controlled by a number of global system variables, which are described in the following sections.

1.5.1 Pending Deletions

The global variable TEDIT.EXTEND.PENDING.DELETE determines whether or not extending a selection makes it a pending-delete selection. Initially, its value is T.

1.5.2 Default Format Specification

The global variable TEDIT.DEFAULT.FMTSPEC contains the default paragraph looks specification. Its initial value is defined by:

TEdit

```
(create FMTSPEC
        QUAD <- (QUOTE LEFT)
        1STLEFTMAR <- 0
        LEFTMAR <- 0
        RIGHTMAR <- 0
        LEADBEFORE <- 0
        LEADAFTER <- 0
        LINELEAD <- 0
        TABSPEC <- (CONS NIL NIL)
)
```

1.5.3 The Current Selection

The global variable TEDIT.SELECTION contains the address of the current selection object which was made in any TEdit window. Note that TEdit may operate across several editing windows concurrently. Consider the example:

```
<-(TEDIT.SETSEL mystream 23 27)
{SELECTION}#55,154672
```

```
<-TEDIT.SELECTION
{SELECTION}#55,154672
```

1.5.4 The Current Shift-Selection

The global variable TEDIT.SHIFTEDSELECTION contains the address of the current selection object from any TEdit window which was created via shift-selection. Suppose I shift-select the words "polynomial equation".

```
<-TEDIT.SHIFTEDSELECTION
{SELECTION}#61,12463
```

TEdit

1.5.5 The Current Move Selection

The global variable TEDIT.MOVESELECTION contains the address of the current selection object from any TEdit window which was created via <CTRL>shift-selection.

1.5.6 The Current Read Table

The global variable TEDIT.READTABLE contains the address of the current read table that is used by TEdit. Consider the example:

```
<-TEDIT.READTABLE  
{READTABLEP}#71,42720
```

1.5.7 The Word Boundary Read Table

The global variable TEDIT.WORDBOUND.READTABLE contains the address of the current read table which controls TEdit's concept of word boundaries. Entries in this table specify the syntax classes which are treated as white space. Consider the following example:

```
<-TEDIT.WORDBOUND.REATABLE  
{READTABLEP}#71,42714
```

1.5.8 TEdit Default Properties

The global variable TEDIT.DEFAULT.PROPS has as its value a list, which is the set of default properties for TEdit or OPENTEXTSTREAM. Its initial value is NIL which indicates that the current settings in the system font profile are to be used.

TEdit

If you specify optional properties when you invoke TEdit, these properties are appended to the front of the list so that they override any defaults.

2. Display-Oriented Structure Editor

DEdit is a structure-oriented, display-based editor for editing Interlisp data structures and programs. It incorporates the features of the standard TTY-oriented Interlisp editor. However, it provides a menu- and window-based environment for accessing the operations provided by the standard Editor.

A structure editor operates on pieces of objects as structures in their own right. For example, DEdit can work on a COND expression embedded within an Interlisp function or on an element of a fairly complex list structure. The specification of the piece of structure is made by the user by selecting an expression using the mouse. Thus, an expression may be an entire function, an S-expression within a function, or even just an atom.

DEdit is a modeless editor. You select operations from a convenient menu which is located at one side of the main editing window. Arguments are identified by using the mouse to select the appropriate pieces of structure prior to selecting the operation.

DEdit is the standard editor for all editing operations performed by Interlisp prior to the Lyric Release. This chapter describes most of the features of DEdit. However, to really understand its power, it is best to experiment with all of the operations. You will quickly recognize the analogs to the standard editor operations.

2.1 Invoking DEdit

DEdit may be invoked on a number of different Interlisp objects and data structures. Different function calls are used to

DEdit

invoke DEdit as described in the following sections. The basic function used by all calls to DEdit is DEDITIT, which is described in Section 2.1.5.

2.1.1 Editing Functions

To edit a function definition, you invoke DEdit using the function **DF**, which takes the form:

Function:	DF
# Arguments:	1
Arguments:	1) FN, a function name
Value:	The function name.

DF is an Nlambda, nospread function. When DF is executed, it prompts you to place a window on the screen in which the function definition is displayed. The DEdit menu is displayed to the right hand side of the window. Figure 2.1 depicts the editing of a function definition.

```
<-DF(USABLE.HEIGHT)
USABLE.HEIGHT
```

DEdit

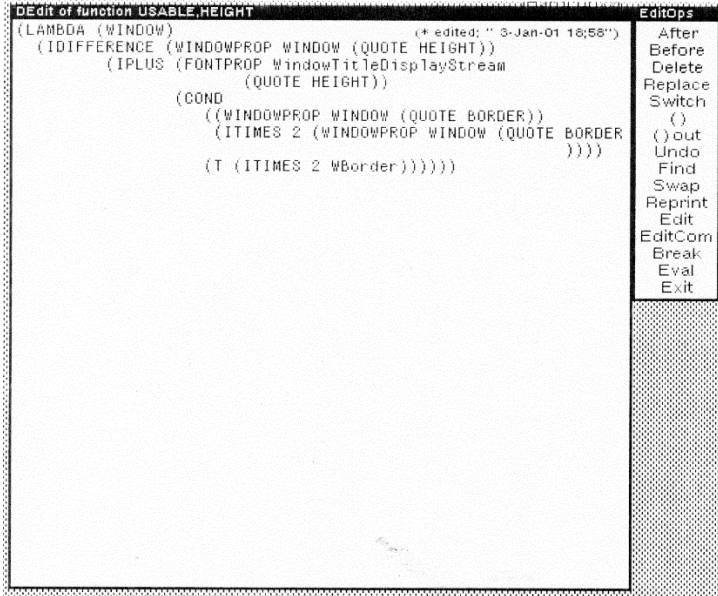


Figure 2-1. Editing a Function Definition

We might define DF as follows:

```
<-(DEFINEQ      (DF (NLAMBDA FN)
                 (DEDITIT 'EDITF FN 'DISPLAY)
                 ))
(DF)
```

2.1.2 Editing Variable Values

You may edit the value of a variable by invoking DEdit using the function **DV**, which takes the form:

Function:	DV
# Arguments:	1

DEdit

Arguments:

1) VAR, a variable name

Value:

The variable name.

DV is an Nlambda, nospread function. When DV is executed, it prompts you to locate a window on the display screen in which the value of the variable is displayed. The DEDIT menu appears on the right hand side of the window. Figure 2.2 depicts the editing of a variable value.

<-DV(INITCOMS)

INITCOMS

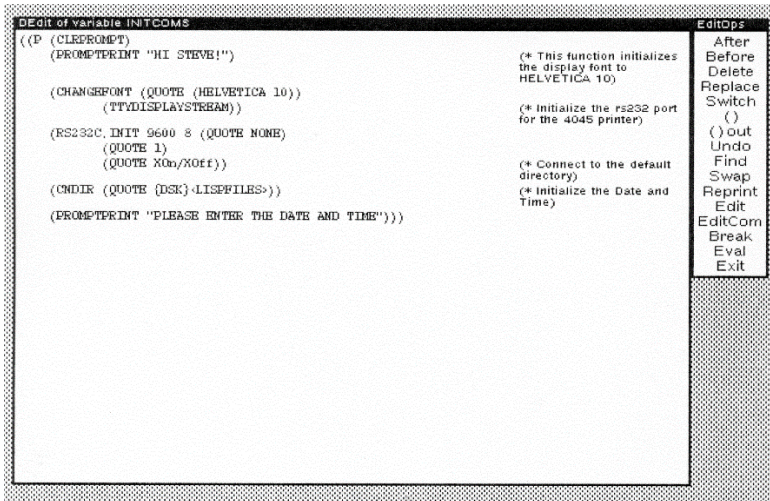


Figure 2-2. Editing a Variable Value

We might DV as follows:

```
<-(DEFINEQ (DV (NLAMBDA VAR
  (DEDITIT 'EDITV VAR 'DISPLAY)
  )))
```

DEdit

(DV)

2.1.3 Editing a Property List

You may edit the value of a property of an atom or the entire property list using the function **DP**, which takes the form:

Function:	DP
# Arguments:	2
Arguments:	1) NAME, the name of an atom 2) PROP, the name of a property
Value:	The property name.

DP is an Nlambda, nospread function. When DP is executed, it prompts you to locate a window on the display screen in which it displays the value of the specified property. The DEdit menu appears on the right hand side of the window.

If PROP has the value NIL, then DEdit displays the entire property list of NAME for you to edit. Figure 2.3 depicts the editing of a property list.

```
<-(DP CAR)  
CAR
```

DEdit

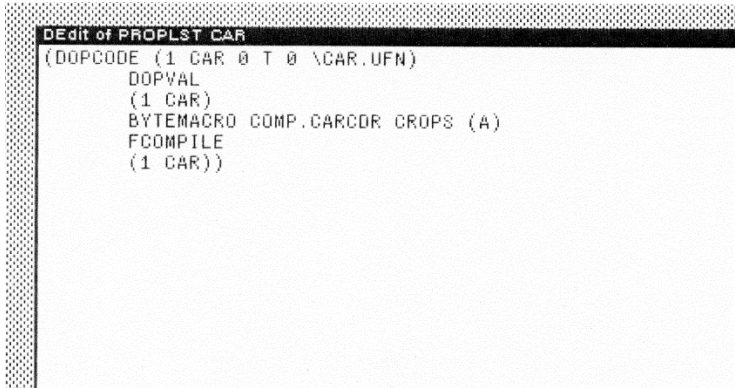


Figure 2-3. Editing of a Property List

We might define DP as follows:

```
<-(DEFINEQ (DP (NLAMBDA ATOM
  (DEDITIT 'EDITPROP (MKLIST ATOM) 'DISPLAY)
  )))
(DP)
```

Note that we make the value of ATOM a list so that we can easily handle both cases for DP.

2.1.4 Editing File Commands

You may edit the File Package commands for a file using the function **DC**, which takes the form:

Function:	DC
# Arguments:	1
Arguments:	1) FILE, the name of a file
Value:	The name of the variable containing the File Package

DEdit

commands.

DC is an Nlambda, nospread function. When DC is invoked, you are prompted to locate a window on the display screen in which the File Package commands (e.g., the value of the variable <FILE>COMS) is displayed. The DEdit menu appears to the right hand side of the window. Figure 2.4 depicts the editing of File Package commands.

```
<-(DC SHKCOMS)
SHKCOMS
```

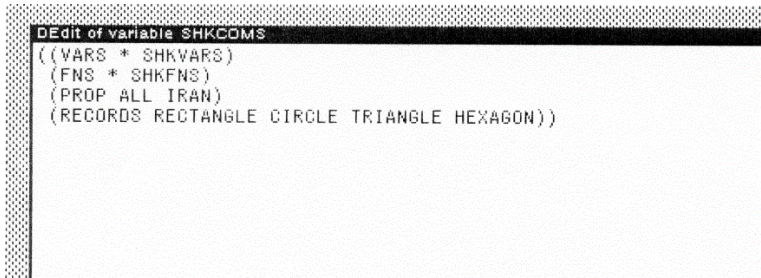


Figure 2-4. Editing File Package Commands

If FILE is not loaded, DC prints a warning message and terminates:

```
<-(DC 'SHK)
(SHK) is not a loaded file.
```

We might define DC as follows:

```
<-(DEFINEQ (DC (NLAMBDA FILE
  (* Isolate the file name)
  (SETQ FILE (OR (CAR (LISTP FILE)) FILE))
  (* Make sure the file has been defined, e.g., there are
```

DEdit

File Package Commands loaded for it.)

```
(DEDITIT 'EDITV
  (if (HASDEF FILE 'FILE NIL T)
    then (FILECOMS FILE)
      else (ERROR FILE "is not a loaded file" T))
  'DISPLAY)
)
))
(DC)
```

2.1.5 The Main DEdit Interface

Each of the functions described above calls a single interface to DEdit. This function is **DEDITIT**, which takes the form:

Function:	DEDITIT
# Arguments:	3
Arguments:	1) EDITFN, an editing function 2) EDITARGS, the expression to be edited 3) EDIT_MODE, the editing mode
Value:	The name of the argument edited.

DEDITIT wraps a call to the function EDITMODE in a RESETFORM in order to ensure that editing of the argument may be restored to its original value if you abort the editing session. The definition of DEDITIT appears as:

```
<-(DEFINEQ (DEDITIT (LAMBDA (EDITFN
  EDITARGS EDITMODE)
    (RESETFORM (EDITMODE EDIT_MODE)
      (APPLY EDITFN EDITARGS))
  )
))
```

DEdit

(DEDITIT)

EDITMODE is used to decide which environment to edit in. This is important if you are transporting a program from a non-windowing environment such as Interlisp-10 to another system running Interlisp. EDITMODE merely decides whether to call the standard Interlisp Editor (e.g., the function EDITL or DEDITL, which is the display-oriented version of the editor.

2.2 DEdit Operation

DEdit's philosophy is to emphasize the use of interactive operations to edit Interlisp objects and data structures. When DEdit is invoked for the first time (in a session), it prompts you for the location of an edit window which you specify by moving the mouse to the appropriate position on the display screen. The edit window is given a fixed size by DEdit when it is initially created. You may reshape the window to suit your needs at any time. The edit window handle is cached in a DEdit system variable for later re-use.

The object or data structure to be edited is displayed in the edit window via the prettyprinter (which, incidentally, ignores the contents of PRETTYPRINTMACROS).

A standard Interlisp scroll bar is attached to the left edge of the window. Of course, the scroll bar is only useful if the size of the window's contents exceeds the available display space.

A standard DEdit menu is attached to the right edge of the window. It remains active throughout the editing process. Figures 2.1 through 2.4 all depict the standard DEdit menu.

DEdit

DEdit operates in an iterative loop consisting of a *select-execute* cycle.

During the editing process, the RIGHT mouse button causes the standard window menu to be displayed when pressed while the cursor is located in the title pane of the editing window.

2.2.1 The Selection Phase

The *selection phase* selects the object or objects to be operated upon in the window. To select an item in the window move the mouse to point directly at the object. Selection is accomplished through judicious use of the three mouse buttons, which has the effect described in Table 2-1.

Table 2-1. Mouse Button Effects

Button	Effect
LEFT	Selects the object that is directly pointed at by the mouse. Figure 2.5 depicts this action. If the object pointed to is a list (indicated by pointing to its opening or closing parenthesis), the entire list is selected.
MIDDLE	Selects the list which contains the item that is currently pointed at by the mouse. Figure 2.6 depicts this action.
RIGHT	Selects the lowest common ancestor of the item and position that are currently pointed at by the mouse. Figure 2.7 depicts this action.

DEdit

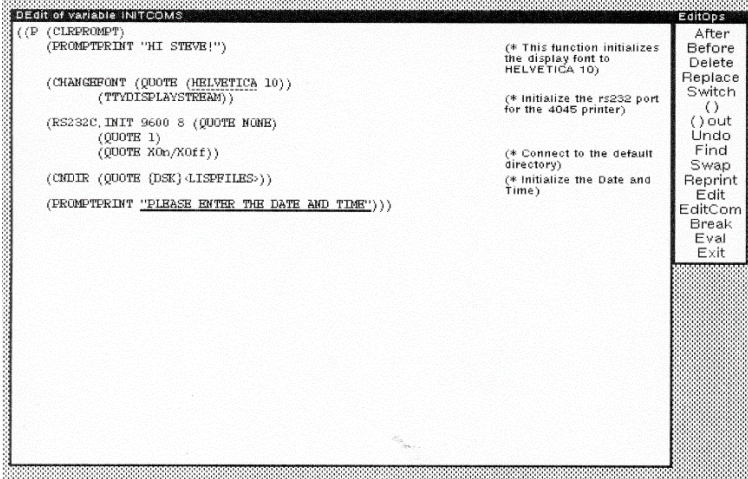


Figure 2-5. Selecting an Item via the LEFT Mouse Button

The only items that you may point at in the editing pane are atomic objects such as atoms, numbers, strings, etc. Pointing at one of a matching pair of parentheses indicates that you want to select the list which they delimit. White space is neither selectable nor editable.

When a selection is made, it is pushed on an internal stack. DEdit commands are applied to the top of the stack. The stack can grow arbitrarily deep, so, at most, only the top two selections are highlighted on the screen because DEdit operators take only two operands. The most [itrecent]it selection is underlined with a solid black line while the next most recent is underlined with a dashed line. The selections may overlap each other, but the method of underscoring makes both selections visible to you.

DEdit

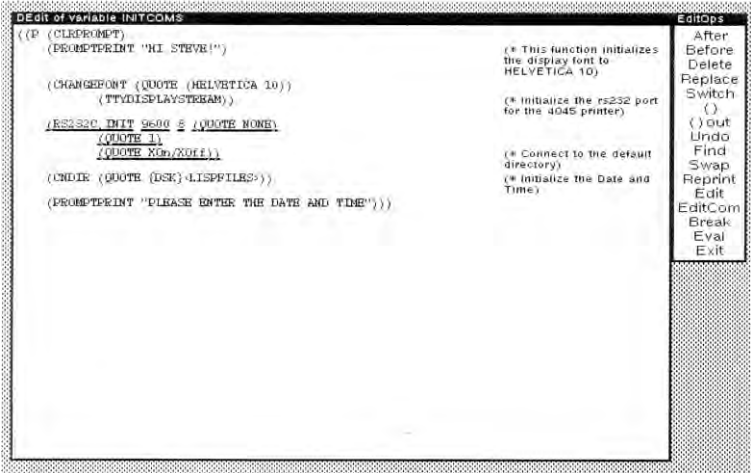


Figure 2-6. Selecting a Containing List

DEdit may be invoked recursively on components of a structure. Thus, several DEdit windows may be open on the display screen at one time. You may make selections in any active DEdit window.

2.2.1.1 Shift-Selection

In many cases, you will want to rearrange pieces of programs or data structures or modify them for inclusion in another part of the program. To avoid the chore of retyping these pieces of text, DEdit supports *shift-selection*. Whenever a selection is made in a DEdit window while holding the left mouse button down, the selected item is not pushed on the stack but is placed in the type-in buffer (see below). Different highlighting is used to indicate that shift-selection has occurred. Figure 2-7 depicts a shift-selection to the type-in buffer.

DEdit

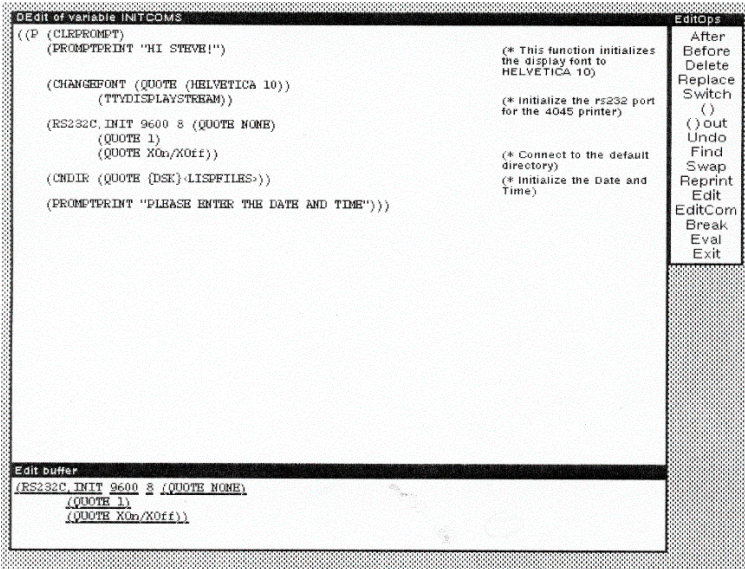


Figure 2-7. Shift-Selection to the Type-in Buffer

Selections copied to the type-in buffer may be edited and then inserted (perhaps, multiple times) into the main editing pane. Selections from the type-in buffer may also be copied to other DEdit windows.

2.2.2 The Execute Phase

The *execute phase* consists of supplying additional operands through type-in as well as selection of a DEdit command to be applied to the specified operands.

2.2.2.1 The Type-in Buffer

For many DEdit commands the operands will be chosen through the selection process outlined above. However, some commands

DEdit

require the entry of new text (such as Insert). During the editing process, a type-in buffer window is attached under the currently active DEdit window (since many DEdit windows may be open at once). You may type characters at the keyboard at any time. They will be entered into the type-in buffer. DEdit reads the characters that you type via LISPXREAD (see Section 25.5.1, I), so the entry process must be terminated by a carriage return or a balancing right parenthesis. During the character entry process you can use the TTYIN character editing subsystem to edit the contents of the type-in buffer.

Once you terminate the type-in process, the type-in structure becomes the top selection on the stack and is available as an operand for the next command selected. The structure displayed in the type-in buffer can be selected from, scrolled (if sufficiently large), or edited just as in the main DEdit window.

2.2.2.2 Type-in Buffer Editing Commands

A few editing commands are available to you directly within the type-in buffer, as depicted in Table 2-2.

Table 2-2. Editing Commands in Type-In Buffer

Command	Effect
CTRL-Z	Interprets the rest of the line as a teletype editor command.
CTRL-S OLD NEW	Substitutes NEW for OLD in the structure.
CTRL-F X	finds the next occurrence of X.

2.2.3 Command Menu

DEdit commands are displayed in a *command menu* that is attached to the right side of the main editing window. You select

DEdit

a command by placing the cursor over the menu item and pressing the LEFT mouse button.

Some commands have a subcommand menu (such as the parenthesis command) which provide frequently used variations of the main command. To access the subcommands, you access the command in the command menu with the MIDDLE mouse button which brings up the subcommand menu.

All commands take their operands from the selection stack. They may push a result back on the stack. The general process for executing a DEdit command is:

1. Select the [ittarget]it structure.
2. Select the [itsource]it structure.
3. Select and execute the command.

The editing process consists of iterating over these three steps until the *Exit* command is selected and executed.

Whenever a command changes the contents of the editing window, the prettyprinter is invoked to reprint the contents of the window.

2.2.3.1 Moving the Command Menu

As you will notice, making selections of program structure and selecting DEdit commands can involve a lot of mouse movement. To avoid the necessity of extraneous mouse movement during a lengthy editing session, DEdit allows you to bring the command menu to the current location of the cursor. To do so, press the TAB key which cause the command menu to *snuggle up* to the cursor. The command menu window will remain at the current location of the cursor as long as the TAB key is held down or the cursor remains within the command

DEdit

menu window. When you release the TAB key, the DEdit Menu returns to its usual position.

2.3 DEdit Commands

This section describes the DEdit commands available through the DEdit command menu attached to the editing window. Subcommands are described along with the main command with which they are associated.

For descriptive purposes, assume that the top two selections on the stack are indicated by the variables TOP and NXT, respectively.

2.3.1 After

After inserts a copy of TOP after NXT in the editing window. If you're target is the entire expression in the edit pane, After will not allow you to insert the new expression.

2.3.2 Before

Before inserts a copy of TOP before NXT in the editing window. If you're target expression is the first expression of the structure to be edited, Before will not allow you to insert the new expression. Note that this prevents you from violating the integrity of functions which are displayed as Lambda expressions.

2.3.3 Delete

Delete deletes TOP from the structure being edited. A copy of TOP remains on the stack and appears in the type-in buffer as a selected item. You cannot delete the entire structure that is

displayed in the edit pane. If you were to do so, this might result in the object being edited receiving the value NIL.

2.3.4 Replace

Replace replaces NXT in the structure being edited with a copy of TOP where NXT appears in the structure. A copy of NXT is substituted into the structure by splicing it in in place of TOP.

2.3.5 Switch

Switch exchanges TOP and NXT in the structure being edited. This operation is most frequently used when you have inverted the ordering of variables although it works equally well for expressions.

2.3.6 Parentheses Insertion

() puts parentheses around TOP and NXT. They may be the same element. This is one of the most useful operations. After editing a piece of the structure, the parentheses may be unbalanced. That is, the new expression is not placed quite right. By judicious insertion (and deletion!) of parentheses, you can assure a proper definition of the value.

() has two subcommands associated with inserting individual left or right parentheses as described in Table 2-3.

Table 2-3. Parenthesis Insertion Subcommands

Subcommand	Effect
(in	Inserts a left parenthesis before TOP. It has the same effect as the LI Editor command.
) in	Inserts a right parenthesis after TOP. It has the same effect as the RI Editor command.

2.3.7 Parenthesis Removal

() out removes the innermost pair of parentheses surrounding TOP. This operation is also very useful in ensuring the proper definition of a structure. For example, if you want to insert a new list into a structure, you must sometimes remove an additional pair of parentheses. **()** has two subcommands associated with removing individual parentheses as described in Table 2-4.

Table 2-4. Parenthesis Removal Subcommands

Subcommand	Effect
(out	Removes the immediate left parenthesis before TOP. It has the same effect as the LO Editor command.
) out	Removes the immediate right parenthesis after TOP. Its effect is exactly like the RO Editor command

2.3.8 Undoing the Previous Command

Undo allows you to undo the last command. It has the same effect as the UNDO Editor command. Undo has three subcommands that allow selective undoing as depicted in Table 2-5.

Table 2-5. Undo Subcommands

Subcommand	Effect
!Undo	Undoes all changes since the start of this call on DEdit.
?Undo	Allows you to selectively undo previous commands from a menu of all command issued since the start of this call on DEdit.

DEdit

&Undo	Allows you to select a command from the menu of all commands issued since the start of this call on DEdit, and undo that command and all commands since.
-------	--

2.3.9 Searching for Structure

Find searches for a piece of structure in the edit window which matches NXT. DEdit looks for the first piece of structure after TOP matching NXT. This command uses all of the capabilities of the Editor's search routines.

2.3.10 Swapping Selections

Swap exchanges TOP and NXT on the stack, but does not effect the contents of the editing window. Swap has a number of subcommands as depicted in Table 2-6.

Table 2-6. Swap Subcommands

Subcommand	Effect
Center	Scrolls the edit window until TOP is visible in the approximate center of the window.
Clear	Discards all selections currently on the stack.
Copy	Copies TOP to the edit buffer and also duplicates it on the stack.
Pop	Pops TOP off the selection stack.

2.3.11 Reprinting a Selection

Reprint reprints TOP by calling the prettyprinter. After several editing commands the structure may have been sufficiently reorganized to be unreadable. Executing Reprint

DEdit

forces the prettyprinter to apply its heuristics to the structure to *beautify* the structure.

2.3.12 Editing a Structure

Edit invokes another version of DEdit on the definition of TOP. If TOP is a list, the CAR of TOP is used. A new DEdit window is opened with the definition of TOP displayed in the editing pane. Note that the DEdit menu is switched to the new edit window as it becomes the active DEdit window.

You may return to a previous DEdit window by placing the cursor in that window and pressing the left mouse button. Notice that the DEdit menu is automatically transferred to the active DEdit window.

DEdit attempts to determine the datatype of the definition of TOP. It uses TYPESOF to find the type of TOP. If more than one datatype exists, you will be prompted via a menu to select the datatype that you wish to edit.

2.3.13 Executing Arbitrary Editor Commands

EditCom allows you to execute an arbitrary Editor command on the structure being edited. The value of TOP is the Editor command to be executed. NXT is the current expression to which the Editor command will be applied. Thus, you select the expression to be edited and then type the Editor command in the Edit buffer.

EditCom makes several of the commonly used Editor commands available through a menu. These commands include ?=, GETD, CL, DW, REPACK, CAP, RAISE, and LOWER.

DEdit

2.3.14 Inserting a Break Around an Expression

Break performs a BREAK AROUND (see Section 20.3.4,I) around the current expression which is the value of TOP.

2.3.15 Evaluating an Expression

Eval evaluates the expression which is the value of TOP. The result of the evaluation becomes the new value of TOP and also appears in the edit buffer. The evaluation occurs within the context of the structure that is being edited.

2.3.16 Exiting from DEdit

Exit allows you to exit from the current DEdit session. The current structure is saved as the value of the object being edited. Exit has two subcommands available through a menu as depicted in Table 2-7.

Table 2-7. Exit Subcommands

Subcommand	Effect
Ok	Exits without an error.
Stop	Exits with an error.

2.3.17 DEdit Command List

The DEdit command list is stored as the value of the variable **DEDITCOMS**. Its initial value is:

((After DEDITAfter)
(Before DEDITBefore)
(Delete DEDITDelete)
(Replace DEDITReplace)

DEdit

```
(Switch DEDITSwitch)
("()" DEDITBI
  ("()" in" DEDITBI)
  (" in" DEDITLI)
  (") in" DEDITRI))
("() out" DEDITBO
  ("() out" DEDITBO)
  (" out" DEDITLO)
  (") out" DEDITRO))
(Undo DEDITUndo
  (Undo DEDITUndo)
  (!Undo (DEDITUndo T))
  (?Undo (UNDOCHOOSE))
  (&UNDO (UNDOCHOOSE T)))
(Find DEDITFind)
(Swap DEDITSwap
  (Center DEDITCenter)
  (Clear (SETQ \DEDITSELECTIONS NIL))
  (Copy DEDITCopy)
  (Pop (POPSELECTION))
  (Swap DEDITSwap))
(Reprint DEDITReprint)
(Edit DEDITEdit
  (Dedit% Def (DEDITEdit 'DISPLAY 'Def))
  (Dedit% Form (DEDITEdit 'DISPLAY 'Form))
  (TTYEdit% Def (DEDITEdit 'TELETYPE 'Def))
  (TTYEdit% Form (DEDITEdit 'TELETYPE 'Form))
  (TTYIn% Def (DEDITEdit 'TTYIn 'Def))
  (TTYIn% Form (DEDITEdit 'TTYIn 'Form)))
(Editcom DEDITEditCom (?= DEDITARGS)
  (GETD (DEDITEditCom 'GETD))
  (CL (DEDITEditCom 'CL))
  (DW (DEDITEditCom 'DW))
  (REPACK (DEDITEditCom 'REPACK))
  (CAP (DEDITEditCom 'CAP))
```

DEdit

```
(LOWER (DEDITeditCom 'LOWER))  
(RAISE (DEDITeditCom 'RAISE)))  
(Break DEDITBreak)  
(Eval DEDITEval)  
(Exit DEDITExit  
  (OK DEDITExit)  
  (STOP (DEDITExit T)))  
)
```

Chapter Three

The Typein Editor (TTYIN)

Interlisp provides a powerful editor for reading input from the terminal (i.e., the keyboard) and editing it prior to passing it to the requesting function. This editor is called TTYIN (for TTY Input - an anachronism). An earlier version was made available in Interlisp-10 as a Lispuser's Package, but an enhanced TTYIN has been integrated into the Interlisp system.

TTYIN supports the following features:

- alternate mode completion
- spelling correction
- a help facility
- flexible editing

TTYIN may be used in two ways:

1. You can make it your primary input interface by setting the value of LISPXREADFN to TTYIN, so that the LISPX executive will use it to obtain input.
2. You can call TTYIN directly from within your program whenever it requires input.

The sections of this chapter discuss how to use TTYIN from both the user's and the programmer's perspective. In most cases the user should be made aware that he or she is talking to TTYIN so that they may take advantage of its capabilities.

3.1 Using the Mouse with TTYIN

You may use the mouse when you are interacting with TTYIN. Its three buttons will be interpreted as shown in Table 3-1.

Table 3-1. TTYIN Mouse Button Usage

Button	Effect
LEFT	Moves the caret (i.e., the edit marker) to where the cursor is pointing. The caret moves while you hold down the left button. At the point at which you let up on the mouse button, new text can be entered.
MIDDLE	Moves the caret to the next word boundary closest to the cursor.
RIGHT	Deletes text from the cursor to the caret in either the forward or backward direction. While holding the RIGHT button, sweep the mouse over the text to be deleted, which will be highlighted. When you release the right button, the text is erased. To cancel the action of this button, move the mouse outside the scope of the text block and release the right button.

3.1.1 Secondary Mouse Operations

A secondary set of mouse operations is available when you simultaneously hold down the CTRL and/or SHIFT keys while pressing a mouse button. The operations are similar, but are activated in a different manner. While holding down one of the keys mentioned above, the mouse buttons operate as presented in Table 3-2.

Table 3-2. TTYIN Secondary Mouse Button Usage

Button	Effect
LEFT	Selects a character.
MIDDLE	Selects a word.
RIGHT	Allows you to extend the selection either left or right by sweeping the mouse.

In this mode the caret does not move, but the selected text is highlighted to indicate the type of operation to be performed. When you have made a selection and released all mouse buttons, you may release the key and the operation will be performed. The operations and highlighting corresponding to the keys are presented in Table 3-3.

Table 3-3. Selection Functions

Key	Operation	Highlighting
SHIFT	The selected text is used as typein at the caret.	broken underline
CTRL	The selected text is deleted.	complemented
CTRL-SHIFT	The selected text is deleted and inserted at the caret as typein.	broken underline and complemented

If you want to cancel the selection at any time, press the LEFT or MIDDLE mouse button and move the cursor outside the text area.

You can retrieve the most recent text which you deleted by pressing the MIDDLE mouse button and typing a blank. The deleted text will reappear at the current location of the cursor.

3.2 Invoking TTYIN

TTYIN is implemented as an Interlisp function which may be called from your program. It takes the following form:

Function:	TTYIN
# Arguments:	8
Arguments:	1) PROMPT, the prompt string 2) SPLST, a spelling list 3) HELP, enables a help facility 4) OPTIONS, an options list 5) ECHOTOFILE, an echo flag 6) TABS, a list of tab stops 7) UNREADBUF, for preloading the buffer 8) RDTBL, a readtable
Value:	An edited version of what is in the buffer.

When TTYIN is called, it prints the value of PROMPT, which may be an atom or a string, and waits for you to type in input. When input is completed, TTYIN returns a list consisting of all of the atoms you have typed in up to the termination character (as indicated by the read table).

OPTIONS, if non-NIL, is used to manipulate the text in the buffer before returning it to the requesting function.

If the global variable TYPEAHEADFLG is T (or non-NIL), TTYIN will permit you to type ahead. Otherwise, it will clear the input buffer before issuing the prompt.

The following sections discuss the effect of the various arguments and present examples of how they are used.

3.2.1 Prompt Characters

DEdit

You may specify your own prompt character to indicate TTYIN input mode. If PROMPT is NIL, then the value of DEFAULTPROMPT will be used. DEFAULTPROMPT has an initial value of "*** ". Consider the following expression:

```
<-(TTYIN)
** HELLO<CR>
(HELLO)
```

If PROMPT is T, no prompt is given to the user. Consider the following example:

```
<-(TTYIN T)
HELLO<CR>
(HELLO)
```

If PROMPT is a dotted pair (i.e., (<prompt1> . <prompt2>)), the CAR is used to prompt for the first line of multiline text, while the CADR is used to prompt for subsequent lines up to a termination character. Either <prompt1> or <prompt2> may be NIL to indicate the absence of a prompt. Consider the following example:

```
<-(TTYIN (CONS '1> '+>))
1> ... first line of text here
+> ... next line of text here
...
+> and so on<CR>
```

Note that <CR> is a termination character for TTYIN.

In many situations, it will probably be easier to rebind DEFAULTPROMPT when you enter a subsystem than to recall TTYIN each time you want to specify a new prompt character.

3.2.2 Spelling Lists

You may specify a spelling list to be used by TTYIN to check the correct spelling of user input or perform word completion. The spelling list consists of a list of dotted pairs of the form (<synonym> . <root>). The spelling list may be one of the Interlisp spelling lists or one that you maintain for your application.

While typing a word to TTYIN, you may press the ESCAPE key. When you do so, TTYIN attempts to complete the spelling of the word by inspecting the spelling list and entering the remaining characters in the buffer. Consider the following examples:

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY))
**L<ESC>IBYA
(LIBYA)
```

where the letters IBYA following the <ESC> were supplied from the spelling list by TTYIN.

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY))
** EGIPT<CR>
=EGYPT
(EGYPT)
```

If the spelling list is a system spelling list, words that are completed by pressing ESCAPE are moved to the front of the list to indicate high interest. If spelling correction cannot be performed, TTYIN flashes the screen to alert you to this situation.

3.2.3 Help Facility

If HELP is non-NIL, the value of this argument determines what happens when you type ? or HELP. If HELP is T, TTYIN prints the value of SPLST for you in a suitable format. For example,

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) T)
** ?
Please select from among: IRAN, EGYPT, LIBYA,
HUNGARY, or other
**
```

If HELP is any other atom or a string containing no spaces, TTYIN executes the following expression: (DISPLAYHELP help).

If HELP is any other Interlisp object, it is printed as is.

If HELP is NIL, ? and HELP are treated as any other input you may type to the program. That is, they are afforded no special treatment.

3.2.3.1 Writing a Help Function

DISPLAYHELP is a user-written function that processes a help request. It takes the form:

Function:	DISPLAYHELP
# Arguments:	1
Arguments:	1) the value of HELP
Value:	Any value, but it is ignored.

DEdit

Initially, DISPLAYHELP is treated as a dummy function which merely returns NIL when called. To make it do something useful, you must provide a function that meets your specific needs. A generic form of DISPLAYHELP might look like:

```
(DEFINEQ (DISPLAYHELP (help)
  (COND
    ((NULL help) NIL)
    ((GETD help)
     (* If its a function, evaluate it!)
     (EVAL help))
    ((ATOM help)
     (SELECTQ help
       (<option 1>
        ...
        (<option N>))))
    (T
     (PRIN1 help))
  )
))
```

The argument you pass to DISPLAYHELP determines what action it takes. If HELP is an atom, then DISPLAYHELP selects the appropriate processing for that atom. This is useful when you have several options that you can supply.

3.2.4 The Options List

TTYIN accepts many options which can affect its behavior. Each of these options is discussed in the following sections along with an example of how it might be used.

3.2.4.1 No Spelling Correction

If you specify the atom NOFIXSPELL in the option list, the spelling list will be used only for the Help facility and Escape completion. TTYIN will not call FIXSPELL on unrecognizable arguments. Usually, you will specify NOFIXSPELL if you plan to handle spelling correction through other mechanisms such as the Help facility. Consider the following example:

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) NIL
'(NOFIXSPELL))
** EGIPT<CR>
(EGIPT)
```

Here, EGYPT is misspelled and not corrected!

3.2.4.2 Spelling Correction Confirmation

If you specify the atom MUSTAPPROVE in the options list, TTYIN attempts to perform spelling correction via FIXSPELL, but requests your approval before actually making any corrections. Consider the following example:

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) NIL
'(MUSTAPPROVE))
** EGIPT<CR>
EGIPT = EGYPT? Yes
(EGYPT)
```

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) NIL
'(MUSTAPPROVE))
** HANGRY
(HANGRY)
```

Here HANGRY is not corrected because it diverges too far (according to the spelling correction algorithm) from any of the candidate words.

3.2.4.3 Autocompletion

If you type a <CR> which leaves a word incomplete, TTYIN will attempt autocompletion using the spelling list if the word CRCOMPLETE is present in the options list. TTYIN inspects the spelling list for a unique entry which may be used to complete the word in the input buffer. Upon completion of the atom, you may continue to type new words. Consider the following example:

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) NIL
'(CRCOMPLETE))
**EG<CR>YPT
(EGYPT)
```

where the letters after the <CR> were supplied by TTYIN from the spelling list.

```
<-(TTYIN NIL '(IRAN EGYPT IRAQ LIBYA HUNGARY)
NIL '(CRCOMPLETE))
** IR<CR>
(IR)
```

If there is no unique entry when CRCOMPLETE is specified, TTYIN merely returns the characters that you typed.

3.2.4.4 Directory Name Completion

If SPLST is NIL and the atom DIRECTORY appears in the options list, TTYIN will interpret an <ESC> as a signal to attempt to complete a directory name.

NOTE: This feature was available only in Interlisp-10, but not Interlisp-D. We will consider making it available in Medley-Interlisp.

3.2.4.5 Username Completion

If SPLST is NIL and the atom USER appears in the options list, TTYIN will interpret an <ESC> as a signal to attempt to complete a username.

NOTE: This feature was available only in Interlisp-10 running under the TENEX operating system, but not in Interlisp-D. We will consider making it available in Medley-Interlisp.

3.2.4.6 Filename Completion

If SPLST is NIL and the atom FILE appears in the options list, TTYIN interprets an <ESC> as a signal to attempt filename completion.

NOTE: This feature is available only in Interlisp-10 running under the TOPS-20 operating system, but not Interlisp-D. We will consider making it available under Medley-Interlisp.

3.2.4.7 Fixing Errors

If the atom FIX appears in the options list, TTYIN examines the word that you have typed. If the word is not in the spelling list or does not correct to an entry in the spelling list, TTYIN interacts with you until an acceptable response is achieved.

```
<-(TTYIN NIL '(IRAN EQYPT IRAQ LIBYA HUNGARY) T
'(FIX))
** IRAD
IRAD?
```

Please select from among IRAN, EGYPT, IRAQ, LIBYA,
HUNGARY

** IRAN
(IRAN)

A blank line (e.g., returning NIL) is always an acceptable response. Consider the following example:

```
<-(TTYIN NIL '(IRAN EQYPT IRAQ LIBYA HUNGARY) T
'(FIX))
```

```
** IRAD
IRAD?
```

Please select from among IRAN, EGYPT, IRAQ, LIBYA,
HUNGARY

```
** <CR>
NIL
```

You may wish to accept responses which are not contained within the spelling list. If so, then you are advised to use one of the options NOFIXSPELL, MUSTAPPROVE, or CRCOMPLETE so that TTYIN does not inadvertently correct your response into something else.

3.2.4.8 Reading Input as a String

If you specify the atom STRING in the options list, the input line is read as a string rather than a list of atoms and treated as free text. Consider the following example:

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) NIL
'(STRING))
```

```
** HUNGARY
"HUNGARY"
```

3.2.4.9 No Case Conversion

If you specify the atom `NORAISE` in the options list, `TTYIN` will not convert any lower case letters into upper case. Consider the following example:

```
<-(TTYIN NIL NIL T '(NORAISE))
** iran<CR>
(iran)
```

But, if the spelling list choices are specified in upper case, Interlisp returns the upper case value corresponding to the item you select. Consider the following example:

```
<-(TTYIN NIL '(IRAN EGYPT LIBYA HUNGARY) T
'(NORAISE))
** iran<CR>                "typed in lower case"
** IRAN                    "Interlisp overwrites in upper case"
(IRAN)
```

Note that Interlisp works primarily with upper case characters. However, comments and other text (such as strings) are often best entered as lower case text.

3.2.4.10 No Value Computation

If you specify the atom `NOVALUE` in the options list, `TTYIN` merely returns `T` if you typed anything in response to its prompt. It returns `NIL` if you typed a blank line.

This option is primarily used with the `ECHOTOFILE` argument to avoid duplicating the input because what you typed has already been copied to the file. This option may also be used when you have typed a particularly long string, such as when you use `TTYIN` for text input, which you do not want duplicated.

3.2.4.11 Multiline Input

If you specify the atom REPEAT in the options list, TTYIN repeatedly prompts you for additional lines until you type a <CTRL>Z. Once you have terminated the input, TTYIN returns one list of all the atoms typed (or a single string if the STRING option is present). Carriage returns will be included in the string as EOL characters. Consider the following example:

```
<-(TTYIN NIL NIL NIL '(REPEAT))
** IRAN<CR>
** EGYPT<CR>
** LIBYA<CR>
** <CTRL>-Z
(IRAN EGYPT LIBYA)
```

3.2.4.12 Reading Textual Input

If you specify the atom TEXT in the options list, TTYIN assumes that REPEAT, NORAISE, and NOVALUE are all true. You may then type continuous free text which is terminated by a <CTRL>Z. Consider the following example:

```
<-(TTYIN NIL NIL T '(TEXT))
** Now is the time for all good men to come to the aid of their
** country, or so it is said.|_^Z
T
```

You may also terminate free text input by typing <CTRL>V. When TTYIN detects a <CTRL>V, it sets the global variable CTRLVFLG and returns.

3.2.4.13 Reading Commands

If you specify the atom `COMMAND` in the options list, `TTYIN` applies spelling correction, if enabled, only to the first word in the list. Thus, `TTYIN` treats the list as if it had the form:

`(<command> . <rest-of-input>)`

The second element is interpreted according to the other options that may be present. Several cases may result as depicted in Table 3-4.

Table 3-4. Reading Commands

Command	Description
<p><code>COMMAND</code> <code>NOVALUE</code></p>	<p>If <code>NOVALUE</code> is specified as an option, <code>TTYIN</code> will return <code>(<command>)</code> if no further input or <code>(<command> . <rest-of-input>)</code> if <code><rest-of-input></code> is non-NIL. Consider the following example:</p> <pre data-bbox="512 852 978 970"><-(TTYIN NIL '(LIST COPY PRINT) NIL '(COMMAND NOVALUE)) ** (LIST <LISPFILES>AC2) (%(LIST <LISPFILES>AC2%))</pre>
<p><code>COMMAND</code> <code>STRING</code></p>	<p>If <code>STRING</code> is specified as an option, <code>TTYIN</code> returns <code><rest-of-input></code> as a string. Consider the following example:</p> <pre data-bbox="512 1098 956 1219"><-(TTYIN NIL "(LIST COPY PRINT) T '(COMMAND STRING)) ** LIST X<CR> (LIST . "X")</pre>
<p><code>COMMAND</code> <code>REPEAT</code></p>	<p>If <code>REPEAT</code> is specified as an option, <code>COMMAND</code> applies only to the first line typed.</p>

3.2.4.14 Using the READ Conventions

Normally, TTYIN will treat parentheses, brackets, and quotes as individual atoms which are loaded into the buffer as you type them. You may force these characters to be treated according to the current readtable specifications by specifying the option READ.

Thus, a balancing ‘)’ or over-balancing ‘]’ will activate the input as will a <CR> when no parenthesis remains unbalanced. Control characters may be entered using the <CTRL>-Vx notation, where x is the control character to be entered.

Note that READ will override all other options except NORAISE.

3.2.4.15 Using LISPXREAD Conventions

As with LISPXREAD, TTYIN will permit you to type-ahead when this option is specified.

3.2.4.16 No Prompting

If NOPROMPT is specified as an option, TTYIN does not print the prompt character for the first line, but assumes that the user program has already done so. However, when echoing the line to a file, the prompt character will be inserted at the beginning of the line.

3.2.5 Echoing Input to a File

If ECHOTOFILE is non-NIL, all user input will be copied to the specified file. For example, specifying T as the value of ECHOTOFILE merely duplicates what you type at your

terminal. If ECHOTOFILE is a list, user input will be copied to all files in the list.

3.2.5.1 Simple Text Entry

You may construct a simple free text entry system by specifying the receiving filename as the value of ECHOTOFILE and specifying NOVALUE in the options list. Consider the following example:

```
<-(OPENFILE 'TEXT 'OUTPUT)
{DSK}<LISPPFILES>TEXT.;1
```

```
<-(TTYIN NIL NIL NIL NIL 'TEXT)
** Now is the time for all good men to come to the aid of their
** country<CR>
T
```

```
<-(CLOSEF 'TEXT)
{DSK}<LISPPFILES>TEXT.;1
```

```
<-(READFILE 'TEXT)
(Now is the time for all good men to come to the aid of their
country)
```

3.2.6 Tabbing Specifications

TABS may be a list of tab stops (e.g., column numbers) which are used for automatic spacing whenever you type a <TAB>. When TTYIN detects a <TAB> in the input, it automatically spaces to the corresponding tab stop in the list. Thus, three <TAB>s entered sequentially will place the cursor at the third tab stop.

3.2.7 Preloading the Input Buffer

You may preload the TTYIN input buffer with a line of input which is specified as the value of UNREADBUF. Essentially, the elements of the list given as the value of UNREADBUF are unread via TTUNREADBUF into the input buffer. Consider the following example:

```
<-(TTYIN NIL NIL NIL NIL NIL "Here is the initial value")
** Here is the initial value
```

and the cursor waits at the end of the line, (e.g., after "value") for you to edit the input. Merely typing a <CR> accepts the contents of the input buffer as the default.

If you type a <CR> (or a <CTRL>Z when REPEAT is specified), the contents of the input buffer will be returned unchanged. Otherwise, you may edit the contents of the buffer according to any options which may have been specified.

When READ is specified as an option, the PRIN2 names of the atoms on the input list will be used.

3.2.8 Read Table Mediation

If RDTBL is non-NIL and a read table, it will mediate the effect of READing input typed by the user. Principally, its effect is to determine how to handle read macros and when to terminate an input line.

This feature is available only in Interlisp. CHECK THIS!!

3.3 TTYIN Editing Commands

TTYIN was developed under Interlisp-10. With the advent of the mouse in Interlisp-D, many of the editing commands will probably not be used because it is easier to position the cursor using the mouse. However, in the interest of providing a complete description, this section will describe all commands for TTYIN.

TTYIN maintains the notion of the *current word* as the word which the cursor currently points to, or if it is a space, the previous word. Parentheses are treated as spaces.

The notation [char] means that you should press the META key and the character simultaneously on your keyboard. The notation \$ indicates the ESCAPE key. Generally, the commands are constrained to one line of text due to their heritage.

Most of the commands may be preceded by a number to indicate how many times the command should be applied. Table 3-5 presents the TTYIN editing commands.

Table 3-5. TTYIN Editing Commands

Command	Explanation
[delete]	Backs the cursor up one or N characters. If the cursor is at the beginning of a line (other than the first line), it deletes the last character of the previous line.
[bs]	Same as above.
[<]	Same as above.
[space]	Move forward one or N characters (if preceded by N).
[>]	Same as above.
[[_^(up-arrow)]	Move up one or N lines (if preceded by N).

DEdit

[lf]	Move down one or N lines (if preceded by N).
[(]	Move backwards one or N words (if preceded by N).
[)]	Move forwards one or N words (if preceded by N).
[tab]	Move to the end of the line.
[\$tab]	Move to the end of the buffer.
[}]	Moves to the end of the buffer.
[CTRL-L]	Moves to the beginning of a line.
[\$CTRL-L]	Moves to the beginning of the buffer.
[{]	Moves to the beginning of the buffer.
[[]	Moves to the beginning of the current list, where the cursor currently resides under an element of the list or its closing parenthesis.
[)]	Moves to the end of the current list, where the cursor currently resides under an element of the list or its beginning parenthesis.
[Sx]	Skips ahead to the next occurrence of the character <i>x</i> or rings the bell if none is found
[Bx]	Searches backward for the previous occurrence of the character <i>x</i> .
[Zx]	Erases all characters from the current position of the cursor to the next occurrence of the character <i>x</i> .
[A]	Repeats the last S, B, or Z command regardless of any intervening input.
[R]	Same as A command.
[K]	Erases the character at which the cursor currently points.
[<CR>]	Restores the buffer's previous contents when it is empty; otherwise, operates just like an [lf] command
[O]	Inserts a <CR><LF> sequence after the cursor but does not move the cursor, which is the same as performing an "open line"
[T]	Transposes the characters before and after the cursor.

DEdit

[G]	Grabs the contents of the previous line from the cursor position onward.
[L]	Sets the rest of the line from the cursor onward to lowercase.
[U]	Sets the current word to uppercase.
[\$U]	Sets the rest of the line from the cursor onward to uppercase
[C]>	Capitalizes the first word of the line.
[CTRL-Q]	Deletes the current line.
[CTRL-W]	Deletes the current word or the previous word if the cursor rests on a space
[J]	Justifies the line according to the line length by moving words from one line to the next and inserting spaces where appropriate.
[\$F]	Finishes the input line by moving to the end of the buffer and inserting a <CR>, CTRL-Z, or] depending on the type of input operation being performed.
[P]	Prettyprints the contents of the buffer in the buffer and inserts appropriate parentheses if required
[N]	Reprints the current line so you can inspect the results of editing.
[\$N]	Reprints the contents of the entire buffer.
[CTRL-Y]	Gets a user exec.
[\$CTRL-Y]	Gets a user exec, but first unreads the contents of the buffer from the cursor to the end of the buffer; this allows you to give something you have typed to Interlisp immediately
[<-]	Adds the current word to the spelling list USERWORDS; with a zero argument, removes the word from the spelling list.
[CTRL-R]	Refreshes the current line. If you type two commands in a row, they cause the entire buffer to be refreshed (on multiline input).

[?]	If typed in the middle of a word, TTYIN attempts to supply alternative completions, if any, from the spelling list.
-----	---

3.4 TTYIN Macros

TTYIN may also be used as character editor from within your own functions. When TTYIN is invoked it opens a window which is attached to the TTYIN buffer. This allows you to dynamically edit an expression from within your program. From within your program TTYIN may be invoked by two macros: ED and EE. Other macros also may load the contents of TTYIN's buffer as described below.

3.4.1 The ED Macro

ED is an Interlisp macro which invokes the TTYIN editor. It loads the current expression into the TTYIN buffer. You may then apply any TTYIN editing command to the expression.

You may exit TTYIN by:

1. Typing a balancing right parenthesis at the end of an expression;
2. Typing <CR> at the end of a balanced expression;
3. Typing <CTRL>-X anywhere in the buffer; or
4. Typing <CTRL>-E to abort.

The ED macro is defined as follows:

```
(ED nil (COMS (TTED)))
```

3.4.2 The EE Macro

DEdit

EE is an Interlisp macro which prettyprints the expression into a buffer. However, it uses its own window to provide an editing area. The EE macro is defined as follows:

```
(EE NIL (COMS (TTED (DO.EE EE))))
```

3.4.3 The BUF macro

The BUF macro loads the current expression into the TTYIN buffer. However, it precedes the expression by E. You may use the contents of the TTYIN buffer as desired. BUF is defined as follows:

```
(BUF  NIL
      (E (LISPXUNREAD
          (TTYIN '* NIL
                NIL
                'LISPXREAD
                NIL
                NIL
                (LIST 'E (##))
          ))
      T)
)
```

Using BUF with no arguments and terminated by a <CR> causes the current expression to be evaluated in a manner similar to that of the Editor macro EVAL.

3.4.4 The TV Macro

The TV macro is a Programmer's Assistant command which operates like EDITV. It allows you to edit the value of a variable by invoking ED on the variable. Consider the following example:

```
<-(TV Actors)
** (IRAN EGYPT LIBYA)<CR>
(not changed)
Actors
```

3.4.5 The FIX Macro

FIX is a Programmer's Assistant command which loads the TTYIN buffer with the specified event's input. This allows you to quickly edit short expressions. Typically, it is used in the Interlisp Executive window to edit an expression typed in at the top-level.

3.5 The ?= Handler

The ?= macro displays the arguments of the function currently being defined in the typein. TTYIN processes this macro internally so that you can continue to edit after you have received the information. When you type ?=, the information is displayed and then the cursor is placed at the location prior to the typing of ?=.

3.5.1 User Handling of ?=

You may provide special treatment for the ?= macro by assigning a value to the variable TTYIN?=FN. Its value should be a function of one argument which is called whenever a ?= is typed in the TTYIN buffer. The argument is the function that ?= is inside of (e.g., you have already typed the function name and one or more arguments of the function).

The function assigned to TTYIN?=FN should return one of the values listed in 3-6.

Table 3-6. ?=FN Values

Value	Meaning
NIL	Indicates that normal ?= processing will be performed.
T	Indicates that nothing should be done. Your function may perform other actions, but nothing is made apparent in the TTYIN buffer.
a list	The list takes the form (ARGS . <STUFF>). It treats <STUFF> as the argument list of the function and performs normal ?= processing, e.g., <STUFF> is displayed in the TTYIN buffer.

If any other value is returned by the function, it is treated as the value to be displayed and is printed in the TTYIN buffer in lieu of the information normally printed by ?=.

3.5.2 Reading Intermediate Arguments

When you type ?=, nothing has been "read" yet by the Interlisp input functions, so you don't have a normal evaluation context in which to examine the values of the arguments. The function assigned to TTYIN?=FN may want to examine the arguments typed after the function name, but before the ?=. It can retrieve them by executing TTYIN.READ?=ARGS, which takes the form:

Function:	TTYIN.READ?=ARGS
# Arguments:	0
Arguments:	N/A
Value:	A list of the arguments typed before the ?=.

DEdit

TTYIN.READ?=ARGS gathers the input between the function name and the ?= into a list and returns it to the calling function. Consider the following example where we define LOOK.AT.ARGS as follows:

```
<-(DEFINEQ (LOOK.AT.ARGS (FN)
  (TTYINEDIT (TTYIN.READ?=ARGS)
    (CREATEW NIL FN))
))
(LOOK.AT.ARGS)

<-(SETQ TTYIN?=FN (FUNCTION LOOK.AT.ARGS))
(TTYIN?=FN reset)
LOOK.AT.ARGS

<-(SQUARE 5 5 ?=
```

At this point, the ?= handler is invoked. It calls the function LOOK.AT.ARGS. LOOK.AT.ARGS opens a window whose title is the function name (e.g., SQUARE). In the edit pane of the window are displayed the arguments. You may then edit the arguments using the mouse. Typing a <CR> completes editing and returns to the window in which you typed the function name. Figure 3.1 depicts the window in which the arguments can be edited.

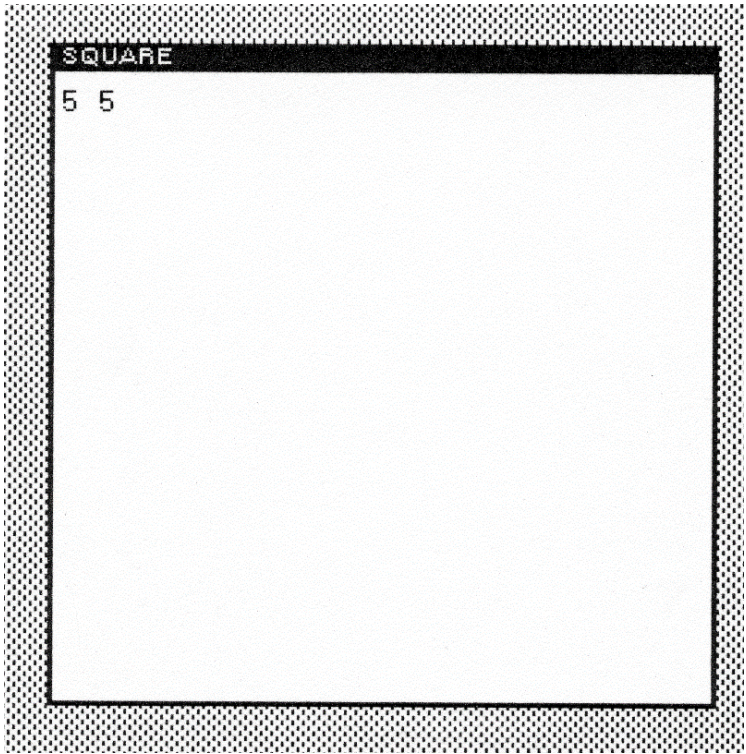


Figure 3-1. Example for LOOK.AT.ARGS

If the `?=` was typed immediately after the function name, `TTYIN.READ?=ARGS` returns `NIL`.

3.5.3 Printing Arguments for `?=`

`TTYIN.PRINTARGS` is the function used by `?=` to print the function and/or arguments. It takes the following form:

Function:	<code>TTYIN.PRINTARGS</code>
# Arguments:	4

DEdit

Arguments:	1) FN, the function name 2) ARGS, an argument list 3) ACTUALS, a list of actual parameters 4) ARGTYPE, a value of function ARGTYPE
Value:	The argument list.

TTYIN.PRINTARGS displays the arguments in the current display stream window.

3.5.4 Enabling ?= Handling

You may enable special ?= handling by setting the value of TTYIN?=FN. If you set its value to NIL, ?= handling is disabled. Otherwise, its argument should be a function of one argument, FN, which is the name of the function in which ?= was typed. You function should return one of the values presented in Table 3-7.

Table 3-7. ?= Handling Values

Value	Meaning
NIL	Normal ‘?’ processing will be performed. Thus, you may intercept ‘?’ and determine if you want to perform any special processing or not.
T	No processing is performed by the system routines. Rather, the assumption is that the user function has done all of the necessary work.
(<arguments> . <list>)	The ‘?’ handler treats <list> as the argument list of the function that you have typed and performs normal ‘?’ processing using it.
<anything else>	The value returned is printed in lieu of the

	normal processing performed by '?='.
--	--------------------------------------

3.6 TTYIN Utility Functions

A number of utility functions are defined that allow you to use the TTYIN processor as an editor from within a program.

3.6.1 Calling TTYIN as an Editor

You may call TTYIN as an Editor by executing the function **TTYINEDIT**, which takes the form:

Function:	TTYINEDIT
# Arguments:	3
Arguments:	1) EXPRS, a list of expressions to be edited 2) WINDOW, a window in which to edit 3) UNPRETTYFLG, a printing flag
Value:	The edited value of EXPRS.

TTYINEDIT is the active function that is invoked by EE. When called, it switches the keyboard and mouse to WINDOW, clears it, and prettyprints EXPRS in the window. At this point, you are executing in TTYIN and may edit the expressions in the window.

If WINDOW is NIL, it uses the value of TTYINEDITWINDOW as the default window in which to edit. If this window does not yet exist (e.g., the first time this function is called), you are prompted to create a window which then becomes the default.

DEdit

If TTYINAUTOCLOSEFLG is non-NIL, the window will be closed automatically when you exit TTYINEDIT.

TTYINEDIT does not prettyprint EXPRS if UNPRETTYFLG is non-NIL.

Consider the following example:

```
<-(TTYINEDIT (LIST MYTEXT) IOWINDOW)
```

```
("A complex number is a generalization of a real number that is  
introduced in mathematics ...)
```

TTYINEDIT acts like a simple text editor. The value of MYTEXT is displayed in the window (see Figure 3.2). You may then edit the value of the expression by positioning the cursor at some point in the expression and inserting/deleting text as previously described in Section 3.1. Note that you exit the window by placing the cursor at the end of the expression and type a <CR>.

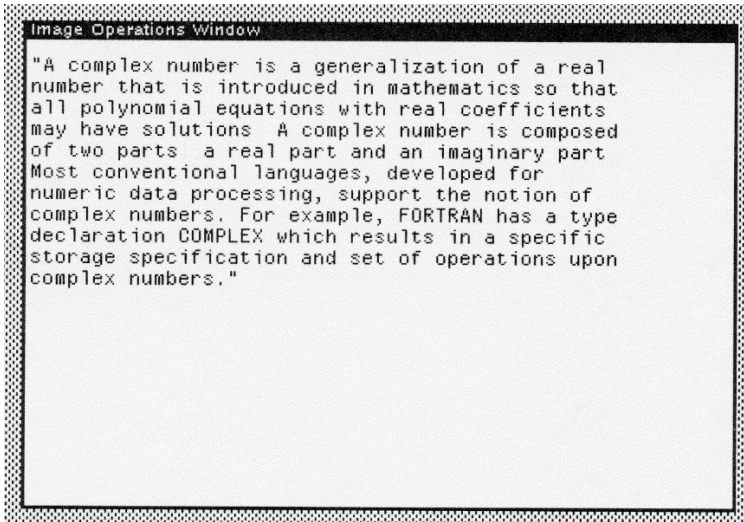


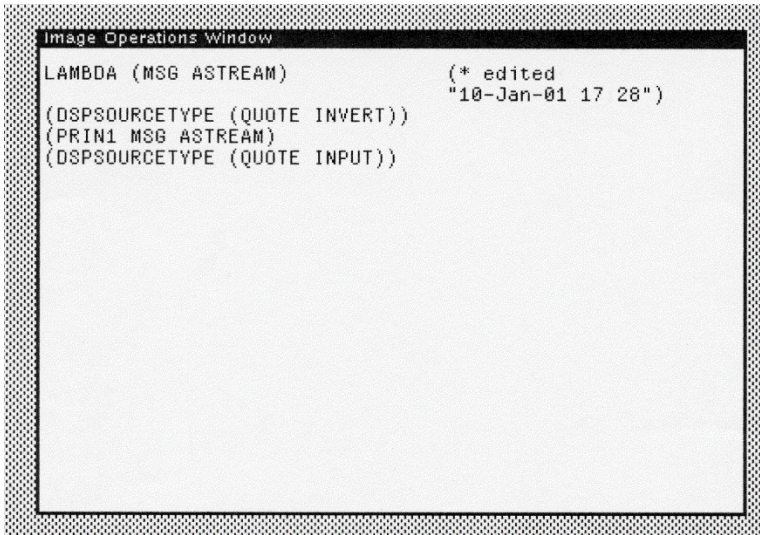
Figure 3-2. TTYINEDIT Editing a String

You may also edit function definitions using TTYINEDIT. Consider this example:

```
<-(TTYINEDIT (GETD 'HIGHLIGHT) IOWINDOW)
(LAMBDA (MSG ASTREAM) (* edited: "10-Jan-01 17:28")
(DSPSOURCETYPE (QUOTE INVERT)) (PRIN1 MSG
ASTREAM) (DSPSOURCETYPE
(QUOTE INPUT)))
```

This is depicted in Figure 3.3.

DEdit

A screenshot of a terminal window titled "Image Operations Window". The window contains the following text:

```
LAMBDA (MSG ASTREAM) (* edited  
"10-Jan-01 17 28")  
(DSPSOURCE (QUOTE INVERT))  
(PRIN1 MSG ASTREAM)  
(DSPSOURCE (QUOTE INPUT))
```

Figure 3-3. TTYINEDIT Editing a Function

You may quickly edit a function definition by invoking TTYINEDIT on its definition.

3.6.2 Setting the TTYIN Window

The function **SET.TTYINEDIT.WINDOW** is used to set the current TTY display stream to the specified window. It takes the form:

Function:	SET.TTYINEDIT.WINDOW
# Arguments:	1
Arguments:	1) WINDOW, a window for editing in TTYIN
Value:	The window handle.

SET.TTYINEDIT.WINDOW is called under a RESETLST. It switches the current TTY display stream to the specified window and clears it. The Y-position in the window is set so that you may then call TTYIN from within your program. Consider the following example:

```
<-(SET.TTYINEDIT.WINDOW IOWINDOW)
{WINDOW}#64,152404
```

You may then enter commands or data in the new window with full access to the TTYIN functions.

3.6.3 Creating a TTYIN Scratch File

The function **TTYIN.SCRATCHFILE** returns the scratch file that is used by TTYIN when prettyprinting its input. It takes the form:

Function:	TTYIN.SCRATCHFILE
# Arguments:	0
Arguments:	N/A
Value:	The name of the scratch file.

If the scratch file does not exist, TTYIN.SCRATCHFILE creates it in the current directory. It sets the file pointer to 0.

You should be careful when creating multiple processes that simultaneously invoke TTYIN because they will all use this file. Thus, it is a good idea to protect calls to TTYIN via monitor locks. Consider the following example:

```
<-(TTYIN.SCRATCHFILE)
{STREAM}#77, 177320
```

This call has opened a stream to the scratch file in the current directory.

3.7 TTYIN Variables

TTYIN's behavior is controlled by a number of variables which may be set by the user.

3.7.1 Automatic Window Closing

When you invoke TTYIN, it opens a window in which you can perform editing (Interlisp only). When TTYIN exits, the window will be closed if the value of the variable TTYINAUTOCLOSEFLG is non-NIL. Its default value is:

```
<-TTYINAUTOCLOSEFLG  
T
```

Typically, you want TTYIN to operate as a "pop-up" editor which appears when needed, but is otherwise unobtrusive. Because the window is cached in TTYIN, it is created only once.

Note that this variable does not affect the action of TTYINEDIT when you use with a window that is already opened.

3.7.2 Allowing Typeahead

If the value of TYPEAHEADFLG is T, TTYIN will allow you to type ahead several commands after you have typed a closing parenthesis or bracket or typed <CTRL>-X. Otherwise, it clears the input buffer of all information except that read by LISPXREAD.

3.7.3 Providing Alternative Completions

If `?ACTIVATEFLG` is T, TTYIN attempts to provide alternative completions from the current spelling list when you type a `?`.

3.7.4 Showing Matching Parentheses

When you are typing a lengthy expression, it is often hard to determine where the matching parenthesis lies to the closing parenthesis that you just typed. If you set `SHOWPARENFLG` to T, whenever you type a right parenthesis or bracket, TTYIN briefly moves the cursor to the matching left parenthesis so that you can establish your position in the expression. The cursor remains at that location for about one second or until you type another character. If you are a fast typer, you probably won't notice the cursor flicking back and forth across the expression.

3.7.5 Errorset Protection

If the value of `TTYINERRORSETFLG` is T, all input not directed to `LISPXREAD` will be errorset-protected. This means that a `<CTRL>-E` will trap back to the prompt character. Otherwise, errors propagate upwards in the control stack. Its value is initially T.

3.7.6 Enabling Escape Completion

If the value of `TTYINCOMPLETEFLG` is T, TTYIN will try to complete a word after you type an escape character (e.g., `<ESC>`) from the `USERWORDS` spelling list.

`USERWORDS` contains words that you have recently mentioned, including the names of functions that you have

defined and variables that you have set or evaluated. Under the theory of temporal locality of reference, if you have mentioned a word recently, there is a great likelihood that you will mention it again. Thus, by typing an escape character (e.g., <ESC> which usually echoes as \$), TTYIN can attempt to complete the word from USERWORDS:

1. If there is no completion, the <ESC> merely echoes as a \$.
2. If there is more than one possible completion, TTYIN signals you with a "beep" to type at least one more character.

If you type an <ESC> while not inside a word (e.g., TTYIN finds a separator to the left), TTYIN assumes the value of LASTWORD, i.e., the last thing that you typed that was recognized by the Programmer's Assistant. You may disable this feature by setting TTYINCOMPLETEFLG to 0.

Because this feature is dependent upon the length of the spelling list(s), as they grow longer, word completion becomes more time consuming. More importantly, if there are too many alternatives for the characters that you have typed, you may get the wrong completion.

Note that the temporary section of USERWORDS, which is #USERWORDS in length contains items unless you explicitly save them in the permanent section. Thus, it is possible for words to fall off the end of the temporary section and become lost. Then, there will be no completion possibility for certain words.

3.7.7 Caching the TTYIN Edit Window

The TTYIN Edit Window is cached in the variable TTYINEDITWINDOW. Initially, this variable has the value NIL. If the WINDOW argument to TTYINEDIT is NIL, then the

value of this variable is used. If it is NIL, the user is prompted to create a window whose handle is then saved as the value of this variable.

3.7.8 Default Printing Function

The default printing function for the variable PRINTFN in EE's call to TTYINEDIT is stored in this variable.

3.7.9 Handling Comments

When entering data to TTYIN, you may want the ability to enter comments along with actual data. TTYINCOMMENTCHAR is a character code. When the first character of a new line of input is equivalent to the value of TTYINCOMMENTCHAR, the line is erased from the screen and not seen by any input function. Usually, its value is the character code for ";". If it is NIL, then comments are not handled in any special way. Initially, its value is NIL.

If you are using TTYIN for copying textual input to a file, then you probably want to disable this feature so that all lines that you type at the keyboard are sent to the file.

3.8 TTYIN READ Macros

When reading input in Interlisp, it is useful to be able to perform translation of special characters to other characters or sequences of code. Read macros provide alternative interpretations of characters as they are read. You may also define read macros for TTYIN that alter the interpretation of characters as they are read by TTYIN.

DEdit

TTYINREADMACROS is a list of entries, each an association list, which defines the behavior to be performed when the specified character is typed. Each entry takes the form:

(<character-code> . <synonym>)

The entries are interpreted as described in Table 3-8.

Table 3-8. TTYIN Read Macro Codes

Code	Interpretation
0Q	The character is ignored.
200Q	Read another character and turn on its edit bit (200Q).
400Q	Read another character and use its original meaning (macro quote).

You may supply more powerful macros which are recognized when a character is typed on an empty line, i.e., as the first character after the prompt. These macros are defined by the form:

(<character-code> <condition> . <response>)

where <condition> is an expression that evaluates true. That is, the response is executed only if the <condition> is true. If <response> is a list, it is evaluated; otherwise, it is not. The result of evaluating (or not) <response> is interpreted as described in Table 3-9.

Table 3-9. TTYIN Read Macro Response

Response	Result
NIL	The macro is ignored and the character is interpreted in a normal manner.

DEdit

<integer>	A character code which replaces the type character. If the value is -1, it is treated as 0 but causes TTYIN to refresh the display of the expression being entered or edited.
<anything>	The TTYIN input is terminated (with a <CR><LF>) and the value of <response> is returned. Thus, you can compute a response which is returned whenever a character is typed. The original character that you typed is not echoed.

Interrupt characters may not be read macros because TTYIN will never see them; they are handled by the low-level I/O routines built into the microcode of the 11xx workstations.

Note that you may have macros which work in a conditional manner by returning either NIL or a non-NIL value. Because you can perform extensive computations "behind the scenes" when a character is typed, you can define macro characters which are context dependent.

New editing commands may be added to TTYINREADMACROS using the following expression:

```
(ADDTOVARS      TTYINREADMACROS
                  (<character-code>
                  'CHARMACRO?
                  <edit-command>)
)
```

3.9 Special Responses

You may specify special processing for a set of commands which are intercepted during any call to TTYIN. The variable TTYINRESPONSES defines these special commands. Its value is a list of elements of the form:

DEdit

(<commands> <response> <option>)

where

- <commands>* is a single atom or list of atoms to be recognized as commands.
- <response>* is an expression to be evaluated (if a list) or applied (if an atom) to the command and the remainder of the line.
- <option>* specifies how the rest of the line is treated.

The function or expression which is the value of *<response>* should return some value which is inserted on the line. If it returns the atom IGNORE, it is not treated as a special response.

There are two free variables defined within TTYIN for this special command processing:

- COMMAND The command that you typed.
- LINE The remainder of the line.

You may reference these free variables in the value of *<response>*, e.g. either a function call, lambda expression, or function name which is applied.

<option> takes one of three values:

1. If it is the atom LINE, the remainder of the line is passed as a list to the function in *<response>*.
2. If it is the atom STRING, the remainder of the line is passed as a string.
3. Otherwise, it should be NIL; the command is only valid if there is nothing else on the line

4. Display Oriented Tools

Many of the Interlisp tools and subsystems were enhanced to operate in the window display system environment. New tools were developed. These are described in the Lisp Library Reference Manual. Additional tools, developed and contributed by Interlisp Users, are described in the Lisp Users Reference manual. Limitations on space prevent us from describing but a few of the display-oriented tools. This chapter discusses some of the tools that are easily and readily used by Interlisp programmers and users.

4.1 The Interactive Display-Oriented Break Package

As described in Volume I: *Interlisp: The Language and Its Usage*, the Break Package was a subsystem oriented to assisting you in debugging your functions and programs through access to the runtime environment. It provided the ability to interactive display the state of the stack, the function code, and the values of data structures.

In Medley Interlisp, the utility of the Break Package is greatly enhanced by providing access to it through the Window Manager. All of the functions available in the TTY-oriented version of the Break Package are available here as well. However, multiple windows allow you to look at many functions, areas of the stack, and data structures simultaneously. This allows you more flexibility in examining the execution and operation of individual functions as well as the program as a whole.

This section will discuss only those features which differ from the TTY-oriented version of the Break Package. Refer to Chapter 20 of Volume I for a discussion of the basic commands.

DISPLAY-ORIENTED TOOLS

4.1.1 Invoking the Display Oriented Break Package

Whenever a break occurs in Interlisp, a break window (see Figure 4-1) is activated near the current tty window of the process in which the break occurs. The current terminal display stream is switched to the trace window. At this point you may type in any of the commands described in Chapter 20 of Volume I. The title of the break window is changed to reflect the name of the broken function, the reason for the break, and the depth of the break recursion. If a break occurs under a previous break, a new break window is created.

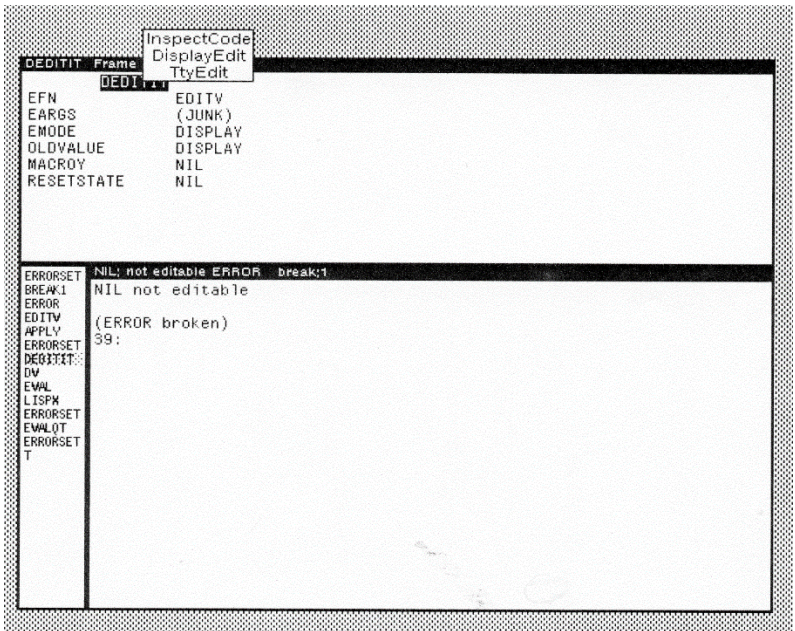


Figure 4-1 The Break Window

DISPLAY-ORIENTED TOOLS

The Display-Oriented Break Package enables a trace window and as many break windows as you find necessary to analyze and debug your program. The break window has the following structure:

- a. A title giving the name of the broken function and the reason for the break.
- b. A display pane in which commands may be typed and results viewed.
- c. A menu of stack frames, which may be scrollable, leading from the expression which failed to the top level.

While the cursor is in the break window display pane, the middle mouse button activates the pop-up menu containing the common Break Package commands (see Figure 4-2).



Figure 4-2 Display-Oriented Break Package Command Menu

Figure 4-3 depicts a window which allows you to inspect a stack frame. In this frame the arguments and their values for the function DEDITIT are displayed. By selecting the name of the stack frame

DISPLAY-ORIENTED TOOLS

with the left mouse button (which highlights it) and pressing the middle mouse button, a pop-up menu of stack frame commands appears (see Figure 4-4).

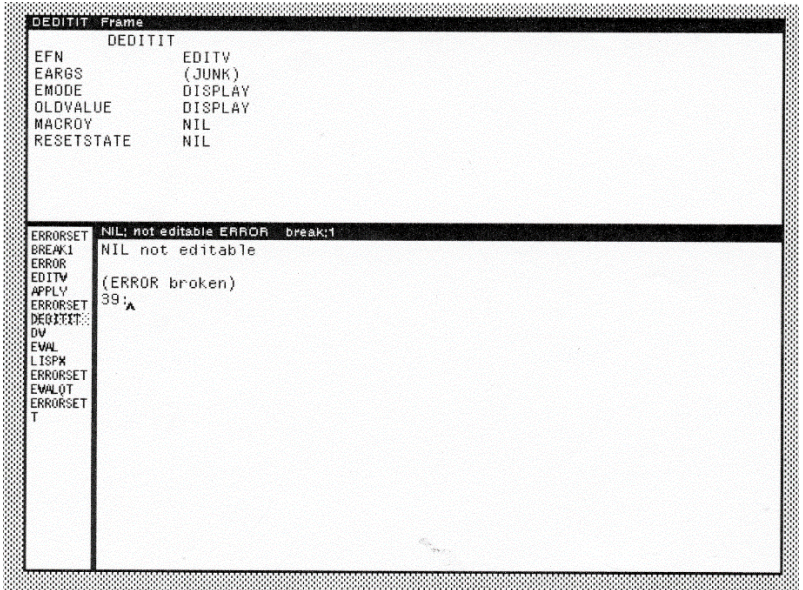


Figure 4-3 Inspecting a Stack Frame

DISPLAY-ORIENTED TOOLS

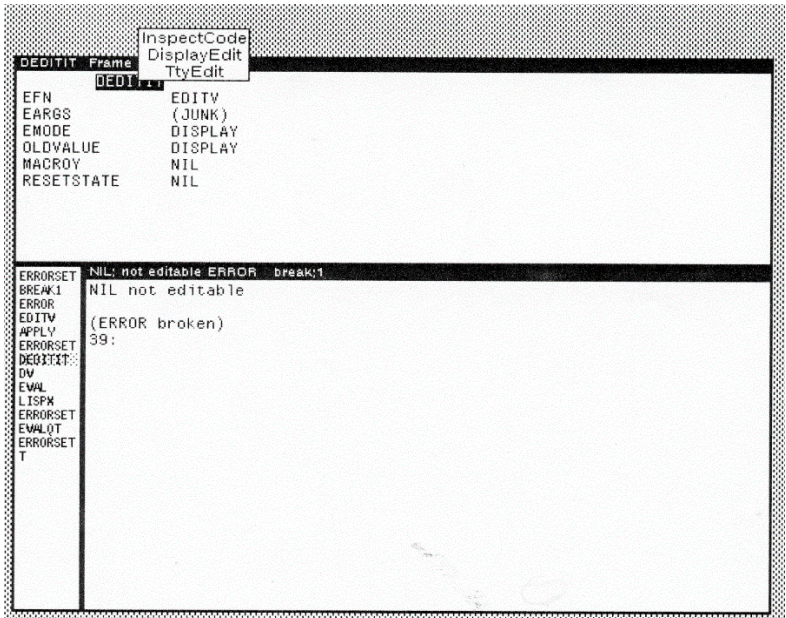


Figure 4-4 Stack Frame Inspection Commands

If you select one of the argument names, such as EMODE, with the left mouse button, it is highlighted in reverse video. Pressing the middle mouse button causes a pop-up menu with the entry SET to appear (see Figure 4-5). Selecting this item causes a pane to appear which requests that you type in an expression for evaluation.

DISPLAY-ORIENTED TOOLS

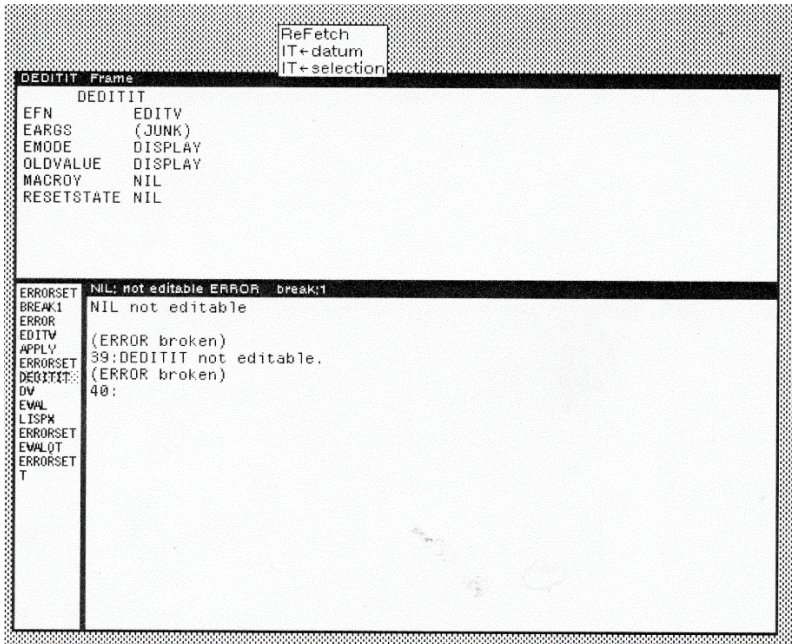
```
Enter the new (EMODE 3) for #71,173422/DEDITIT
The expression read will be EVALuated.
>
DEDITIT Frame
  DEDITIT
  EFN          EDITV
  EARGS        (JUNK)
  EMODE        DISPLAY
  OLDVALUE     DISPLAY
  MACROBY      NIL
  RESETSTATE   NIL

ERRORSET      NIL: not editable ERROR break:1
BREAK1        NIL not editable
ERROR
EDITV         (ERROR broken)
APPLY         39:DEDITIT not editable.
ERRORSET      (ERROR broken)
DEDITIT      48:
DV
EVAL
LISPX
ERRORSET
EVALQT
ERRORSET
T
```

Figure 4-5 Setting EMODE

Placing the cursor in the title bar and pressing the middle mouse button causes a pop-up menu to appear (see Figure 4-6). The primary command of interest is **ReFetch**, which causes the frame arguments to be redisplayed.

DISPLAY-ORIENTED TOOLS



```
Refetch
IT←datum
IT←selection

DEDITIT Frame
DEDITIT
EFN      EDITY
EARGS   (JUNK)
EMODE   DISPLAY
OLDVALUE DISPLAY
MACROY  NIL
RESETSTATE NIL

ERRORSET NIL: not editable ERROR break;t
BREAK1   NIL not editable
ERROR
EDITV   (ERROR broken)
APPLY   39:DEDITIT not editable.
ERRORSET (ERROR broken)
DEDITIT: 40:
DV
EVAL
LISP#
ERRORSET
EVALDT
ERRORSET
T
```

Figure 4-6 Operations on a Stack Frame

4.1.2 Enabling the Display-Oriented Break Package

In the standard Medley Interlisp sysout, the Display-Oriented Break Package is enabled. **BREAK1** has been modified to use the window system to display the break environment. You may enable or disable the Display-Oriented Break Package using the function **WBREAK**, which takes the form:

Function:	WBREAK
# Arguments:	1
Arguments:	1) ONFLG, a flag enabling the Display-Oriented Break Package
Value:	T or NIL.

WBREAK enables or disables the Display-Oriented Break Package. When ONFLG is T, the Display-Oriented Break Package is enabled. Modifications to BREAK1 and other Break Package functions are installed. When ONFLG is NIL, these modifications are removed and the BREAK1 operates like the TTY-oriented version. Consider the following example:

```
<-(WBREAK T)
NIL
```

which enabled the Display-Oriented Break Package.

4.1.3 Display-Oriented Break Package System Variables

The operation of the Display-Oriented Break Package is controlled by a number of variables which are described in the following sections. You should set these variables to suit your preferences in your initialization file (which is loaded by GREET).

4.1.3.1 Back Trace Menu Size

The *backtrace menu* appears (normally) to the right hand side of the break window. It contains a list of the stack frames preceding the frame of the function which broke. If the menu size is too small to contain all of the frames, it is made scrollable in both the vertical and horizontal directions.

The back trace menu size is controlled by the system variables **MaxBkMenuWidth** and **MaxBkMenuHeight** whose default values are 125 and 300 pixels, respectively.

```
<-MaxBkMenuWidth
125
<-MaxBkMenuHeight
```

300

Generally, if you have long function names (over 10 characters), you will want to increase the maximum width of the Break Command Menu. Increasing the height merely allows you to scan more of the calling sequence in a single glance (for deeply nested calling sequences).

4.1.3.2 Performing an Automatic Traceback

When a break occurs, Interlisp will perform an automatic traceback through the stack if AUTOBACKTRACEFLG has a non-NIL value. Its default value is NIL. The difference is whether the system performs the BT command or you do when a break occurs. AUTOBACKTRACEFLG can take one of the non-NIL values depicted in Table 4-1.

Table 4-1. AUTOBACKTRACEFLG Values

Value	Usage
T	On an error, the BT menu is automatically displayed.
BT!	On an error, the BT! menu is automatically displayed.
ALWAYS	The BT menu is displayed on breaks caused both by errors and user-executed invocations of BREAK or <CTRL>B.
ALWAYS!	The BT! is displayed on breaks caused both by errors and user-executed invocations of BREAK or <CTRL>-B.

Generally, you should assign the value ALWAYS to this flag.

4.1.3.3 The Back Trace Font

The back trace font is the font in which the backtrace menu is printed.

```
<-BACKTRACEFONT  
{FONTDESCRIPTOR}#70,171504
```

which corresponds to GACHA 8 MRR.

4.1.3.4 Automatic Closing of the Break Window

When you exit a break, Interlisp will automatically close the break window. However, you may preclude this by setting `CLOSEBREAKWINDOWFLG` to `NIL`. Thereafter, you must close all break windows either via the right button mouse menu or `CLOSEW`.

Initially, this variable has the value `T`. Generally, you should leave it set to `T`. If you have a recursive function that generates several break windows, it can be a rather tedious process cleaning them up.

4.1.3.5 Specifying the Break Window Region

Normally, break windows will be positioned near the window which has the current TTY display stream. The location of the break window is determined by the variable `BREAKREGIONSPEC` whose `LEFT` and `BOTTOM` fields are offset from the left, bottom corner of the window with the current TTY display stream. The `WIDTH` and `HEIGHT` fields determine the size of the break window.

You may specify a fixed break region where you wish the break window to be placed by setting the variable and saving it on a file. This is useful when you are partitioning the screen so that critical windows do not overlap each other.

4.1.3.6 The Trace Window

The trace window, whose window handle is stored in the variable TRACEWINDOW, is used for tracing the execution of functions. It is opened when the first call to TRACE occurs during the execution of a program. It remains open until you explicitly close it. During program execution, you can set TRACEWINDOW to any window handle in order to force trace information to be displayed in that window.

4.1.3.7 Specifying the Trace Window Region

RACEREGION is the initial location of the TRACEWINDOW on the display screen. Its initial value is determined when Medley Interlisp is initialized. You may specify your own location for the trace region by assigning a new region specification to this variable.

4.2 The Inspector

Interlisp provides a display-oriented facility for inspecting the contents of any data structure that can be defined within Interlisp. This facility is known as the *Inspector*. The Inspector provides you with the capability to change the values of any aspects of the data structures that you inspect and to "walk" around complex data structures. Moreover, some data types have multiple aspects that may be inspected. The Inspector provides a menu of the possibilities when examining these objects. This permits you to select the perspective for the data type which best meets your needs.

When the Inspector is invoked, it displays the field names and values of any arbitrary data structure in a window. You are prompted to place the window on the screen. The Inspector adjusts the size of the window to accommodate the size of the data structure. The

DISPLAY-ORIENTED TOOLS

values of fields of a data structure, if sufficiently complex, may also be inspected.

The Inspector is integrated with both the Display-Oriented Break Package and the Interlisp editors.

4.2.1 Invoking the Inspector

There are several ways to invoke the Inspector on an object:

1. Calling `INSPECT` directly
2. Selecting the Inspect command inside an Inspector Window
3. Using `EDITDEF` to edit an object which is not a list

4.2.1.1 Calling the Inspector

You may invoke the Inspector on an arbitrary data structure using the function `INSPECT`, which takes the form:

Function:	<code>INSPECT</code>
# Arguments:	3
Arguments:	1) <code>OBJECT</code> , an arbitrary Interlisp object 2) <code>ASTYPE</code> , the record type of the object, if any 3) <code>WHERE</code> , the location of the inspect window
Value:	The inspect window handle or <code>NIL</code> .

`INSPECT` opens an inspect window and displays the contents of `OBJECT` in the window. Typically, you will call the Inspector with just the value of `OBJECT`.

DISPLAY-ORIENTED TOOLS

The inspection of records and user datatypes is treated differently by the Inspector. `ASTYPE` provides you with the ability to specify how the contents of `OBJECT` will be displayed. If `ASTYPE` has a value, it is assumed to be the record type of the `OBJECT`. In this case the properties of the record will be displayed in the inspect window. However, if `ASTYPE` is `NIL`, the data type of `OBJECT` will be used to determine its property names in the inspect window. Consider the following record definition:

```
<-(INSPECT 'IMAGEOBJ)  
{PROCESS}#71,22100
```

The record definition of `IMAGEOBJ` is displayed in Figure 4-7.

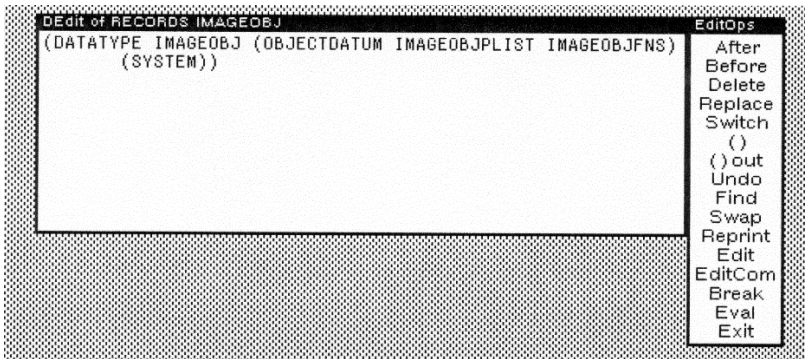


Figure 4-7 Inspection of an `IMAGEOBJ` Record

You may specify the location of the inspect window by assigning a value to `WHERE`. `WHERE` may be an existing window (such as the Interlisp Executive window) or a region which will cause a window to be created in which to display the contents of the data structure.

DISPLAY-ORIENTED TOOLS

WHERE may also be a position whence the lower left corner of the inspect window will be located at those coordinates on the display screen.

4.2.1.2 Inspecting Code

You may inspect the compiled code of a function using the function **INSPECTCODE**, which takes the form:

Function:	INSPECTCODE
# Arguments:	2
Arguments:	1) FN, the name of a function 2) WHERE, the location of the window
Value:	The name of the function.

INSPECTCODE opens a window and displays the compiled code of FN using the function PRINTCODE. If the length of the code exceeds the default size of the window, the window is made scrollable. Consider the following example:

```
<-(INSPECTCODE 'APPEND)
APPEND
```

which creates the window depicted in Figure 4-8.

DISPLAY-ORIENTED TOOLS

```
Code for APPEND
stkmin:      62 na:    -1 pv:    1 startpc:  40
argtype: 2  frame:  APPEND nsize:  0 nlocals:
3
  0:         62
  2:      177777
  4:         1
  6:         40
 10:      200000
 12:      2063
 14:         0
 16:      1400
Local args:
 30:      104      34:         0      IVAR 0: L
 32:         0      36:         0

----
 40:         145      MYARGCOUNT
 41:         21      1  0      BIND      [pvar0];
 44:        110      PVAR      [pvar0]
 45:        144      COPY
 46:         152      '0
 47:        360      EQ
 50:        221      FJUMP-> 53
```

Figure 4-8 Example of INSPECTCODE

WHERE determines the location of the window. It takes the same values as INSPECT above. If NIL, the user is prompted to specify the position and size of the window.

INSPECTCODE invokes TEdit, if loaded, to allow you to edit the code of the function. TEdit is described in Chapter 1.

WARNING! Do not attempt to edit compiled code unless you are thoroughly knowledgeable of the consequences of changes to the code.

If you attempt to apply INSPECTCODE to a function which is not compiled, Interlisp prints a warning and ignores the expression:

DISPLAY-ORIENTED TOOLS

<-(INSPECTCODE 'LOWERLEFT)
LOWERLEFT not compiled code.

4.2.1.3 Inspecting Code from a Break Window

If INSPECTCODE is used to inspect a stack frame name in a Break Window, the location in the code corresponding to the frame's program counter where it was executing at the time the break occurred will be highlighted. Figure 4-9 depicts an example.

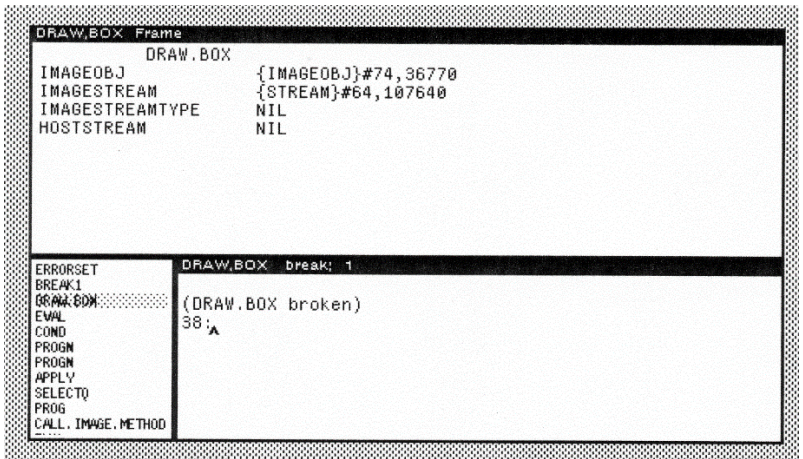


Figure 4-9 Inspecting in a Break Window

Note: If TEdit is loaded, it will be used to create the code inspector window.

4.2.1.4 The Inspect Command

Several of the Interlisp subsystems have the Inspector integrated with them. In each case, an *Inspect* command will be found in a menu associated with those subsystems. When you invoke the Inspector, it determines which aspects apply to the current object

and creates a menu for you to choose from. The most common entries are depicted in Table 4-2.

Table 4-2. Common Menu Entries

Entry	Usage
FNS	Edit the definition of the selected literal atom.
VARs	Inspect a variable value.
PROPS	Inspect the property list of an atom.
Inspect	The object is a list. A window is opened with two columns displayed: a list of numbers and a list of items.
TtyEdit	Invokes TTYIN on the list.
DisplayEdit	Invokes DEdit on a list.
As a PLIST	Inspects a list as if it were a property list (e.g. ((PROPl VALl) ... (PROPn VALn)).
As an ALIST	Inspects the list as an association list (e.g., (PROPl VALl ... PROPn VALn)).
As a record	Displays a submenu of all the records known to the system from which you choose the one whose structure you want to overlay on the list.
As a "record type"	Inspects a list as a record of the type named in its CAR.

If the object that you are inspecting is a bitmap, you will be given a choice of inspecting the bitmap's contents or the bitmap's fields.

4.2.1.5 Editing Variables

If you attempt to edit a non-list data structure via EDITV or DEdit, the Inspector will be called to display the contents of the data structure. Consider the following example:

```
<-(SETQ ABOX (IMAGEOBJCREATE 'BOX BOX))
{IMAGEOBJ}#74,36770
```

```
<-(EDITV ABOX)
ABOX
```

Figure 4-10 depicts the results of editing ABOX.

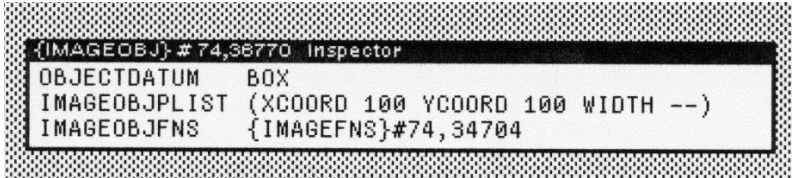


Figure 4-10 Inspecting via EDITV

4.2.1.6 Inspecting Arrays

You may inspect the contents of an array by calling the function **INSPECT/ARRAY**, which takes the form:

Function:	INSPECT/ARRAY
# Arguments:	2
Arguments:	1) ARRAY, an array handle 2) BEGNOFFSET, the element at which to begin inspection
Value:	The window handle.

INSPECT/ARRAY opens a window and displays the contents of ARRAY from BEGNOFFSET through BEGNOFFSET + MAXINSPECTARRAYLEVEL. If BEGNOFFSET is NIL, it defaults to 0. Consider the following example, which is depicted in Figure 4-11:

```
<-(INSPECT/ARRAY \OPCODEARRAY)
{WINDOW}#55,100320
```


DISPLAY-ORIENTED TOOLS

```

{ARRAYP} #71,175340 Inspector
0 (0 -X- 0)
1 (1 CAR 0 T 0 --)
2 (2 CDR 0 T 0 --)
3 (3 LISTP 0 T 0 --)
4 (4 NTPX 0 T 0 --)
5 (5 TYPEP 1 TYPEP 0 --)
6 (6 DTEST 2 ATOM 0 --)
7 (7 CDDR 0 T 0 --)
8 (8 FN0 2 FN 1)
9 (9 FN1 2 FN 0)
10 (10 FN2 2 FN -1)
11 (11 FN3 2 FN -2)
12 (12 FN4 2 FN -3)
13 (13 FNX 3 FNX FNX)
14 (14 APPLYFN 0 T -1)
15 (15 CHECKAPPLY* 0 T 0 --)
16 (16 RETURN 0 T (JUMP 1) --)
17 (17 BIND 2)
18 (18 UNBIND 0)
19 (19 DUNBIND 0)
20 (20 RPLPTR.N 1 T -1 --)
21 (21 GREF 1 T 0 --)
22 (22 ASSOC 0 T -1 --)
23 (23 GVAR+ 2 ATOM 0 --)
24 (24 RPLACA 0 T -1 --)
25 (25 RPLACD 0 T -1 --)
26 (26 CONS 0 T -1 --)
27 (27 GETP 0 T -1 --)
28 (28 FMEMB 0 T -1 --)
29 (29 GETHASH 0 T -1 --)
30 (30 PUTHASH 0 T -2 --)
31 (31 CREATECELL 0 T 0 --)
32 (32 BIN 0 T 0 --)
33 (33 BOUT 0 T -1 --)
34 (34 POPDISP 0 T 0 --)
35 (35 LIST1 0 T 0 --)
36 (36 DOCOLLECT 0 T -1 --)
37 (37 ENDCOLLECT 0 T -1 --)
38 (38 RPLCONS 0 T -1 --)
39 (39 LISTGET 0 T -1 --)
40 (40 ELT 0 T -1 --)
41 (41 NTHCHC 0 T -1 --)
42 (42 SETA 0 T -2 --)
43 (43 RPLCHARCODE 0 T -2 --)
44 (44 EVAL 0 T 0 --)
45 (45 EVALV 0 T 0 --)
46 (46 TYPECHECK 0 T 0 --)
47 (47 STKSCAN 0 T 0 --)
48 (48 BUSBLT 1 (WORDSOUT BYTESOUT BYTE
49 (49 MISC8 1 (IBLT1 IBLT2) -7 --)
50 (50 UREF DAT3 1 (POLY MATRIX 3YS MATE

```

Figure 4-11 INSPECT/ARRAY Example

4.2.2 Inspect Windows

An *inspect window* displays two columns of values. the left column lists the property names of the data structure that you are inspecting. The right column lists the values of the properties given in the left column in a one-for-one correspondence. Figure 4-12 depicts a sample inspect window.

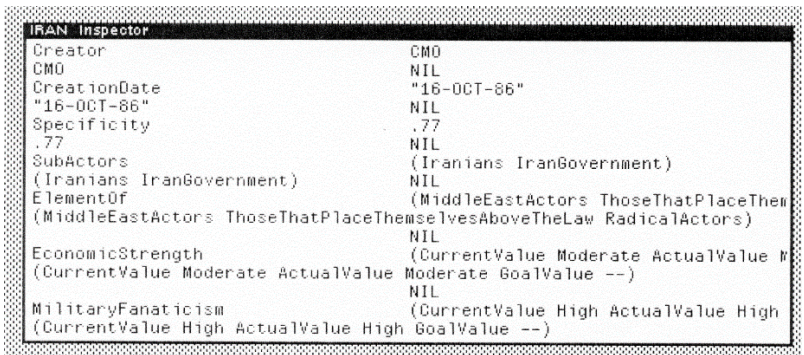


Figure 4-12 A Sample Inspect Window

The left and middle mouse buttons are used to select items in the inspect window and to invoke a menu of commands, respectively.

The left mouse button is used to select items in the inspect window. You may select an item, either a property name or a value, by placing the cursor upon the item and clicking the left mouse button. The item selected will have its shade inverted to indicate that it has been selected. Only one item at a time may be selected in an inspect window. Figure 4-13 depicts a window with a property name selected and the associated menu displayed.

DISPLAY-ORIENTED TOOLS

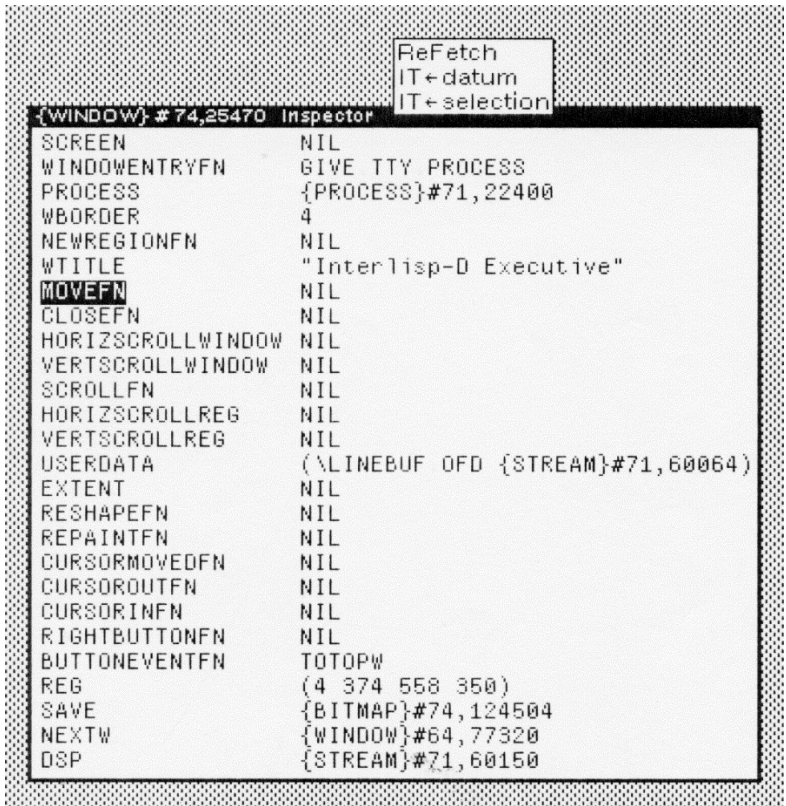


Figure 4-13 Inspector Window with Property Menu

The commands selected by the middle mouse button depend on whether the item selected in the inspect window was a property name or a value. These commands are discussed below. Figure 4-14 depicts an Inspector Window with a property value selected and the associated menu displayed.

DISPLAY-ORIENTED TOOLS

```
{WINDOW}:#74,25470 Inspector
SCREEN                NIL
WINDOWENTRYFN        GIVE.TTY.PROCESS
PROCESS               {PROCESS}:#71,22400
WBORDER               4
NEWREGIONFN          NIL
WTITLE                "Interlisp-D Executive"
MOVEFN                NIL
CLOSEFN               NIL
HORIZSCROLLWINDOW    NIL
VERTSCROLLWINDOW     NIL
SCROLLFN              NIL
HORIZSCROLLREG        NIL
VERTSCROLLREG         NIL
USERDATA              (\LINEBUF.OFD {STREAM}:#71,60064)
EXTENT                NIL
RESHAPEFN             NIL
REPAINTFN             NIL
CURSORMOVEDFN        NIL
CURSOROUTFN          NIL
CURSORINFN            NIL
RIGHTBUTTONFN        NIL
BUTTONEVENTFN        TOTOPW
REG                   (4 374 558 fields
{BITMAP}:#71,22100 contents
SAVE                  {WINDOW}:#64,77320
NEXTW                 {STREAM}:#71,60150
DSP                   {STREAM}:#71,60150
```

Figure 4-14 Inspector Window with Property Value Menu

4.2.3 Creating and Manipulating Inspector Windows

The Inspector is built on an abstraction of a window called an *INSPECTW*. It is a window with certain properties that display objects and respond to the selection of parts of objects. You can create an instance of an *INSPECTW* using the function *INSPECTW.CREATE*, which takes the form:

Function:	<i>INSPECTW.CREATE</i>
# Arguments:	11
Arguments:	1) DATUM, an object handle

DISPLAY-ORIENTED TOOLS

- or address
 - 2) PROPERTIES, a list of a function
 - 3) FETCHFN, a fetch function
 - 4) STOREFN, a storing function
 - 5) PROPCOMMANDFN, a function for the MIDDLE button for property names
 - 6) VALUECOMMANDFN, a function for the MIDDLE button for values
 - 7) TITLECOMMANDFN, a function for the title or border of the Inspect Window
 - 8) TITLE, the title of the Inspect Window
 - 9) SELECTIONFN, a selection function
 - 10) WHERE, the location of the window
 - 11) PROPPRINTFN, a property value printing function
- Value: The window handle.

INSPECTW.CREATE creates an Inspect Window that allows you to view the object DATUM. Consider the following simple example involving a unit from the STRADS program. Let us define the IRAN unit as follows (example formatted for readability):

```
<-(SETQ IRAN NIL)
NIL
<-(PUTPROPS          IRAN
```

DISPLAY-ORIENTED TOOLS

```
Creator CMO
CreationDate "14-OCT-86"
Specificity .77
SubActors (Iranians IranGovernment)
ElementOf
  (MiddleEastActors
   ThoseThatPlaceThemselvesAboveTheLaw
   RadicalActors
   EconomicallyDepressedNations)
EconomicStrength
  (CurrentValue Moderate
   ActualValue Moderate
   GoalValue (>= VeryStrong)
  )
MilitaryFanaticism
  (CurrentValue High
   ActualValue High
   GoalValue (>= High)
  )
MilitaryEnemies
  (CurrentValue (Iraq USA)
   ActualValue (Iraq)
   GoalValue (= NoOne)
  )
)
IRAN

<-(INSPECTW.CREATE 'IRAN
                    (GETPROPLIST 'IRAN)
                    (FUNCTION GETPROP))
{WINDOW}#56,75554
```

which creates the window depicted in Figure 4-12. Note that you must specify the first three arguments to INSPECTW.CREATE in order to generate the window.

4.2.3.1 The Inspect Window Title

TITLE specifies the title of the Inspect Window. Its value is interpreted as follows:

1. If its value is NIL, the title of the window consists of the printed form of the DATUM followed by the string " Inspector".
2. If its value is the literal atom DON'T, the Inspect Window will be displayed without a title.
3. If its value is any other literal atom, it is applied to the datum and the inspect window handle (if known). If the result is the atom DON'T, the Inspect Window will not have a title. Otherwise, the value returned will be used as the title of the Inspect Window.
4. Any other value will be used as the title of the Inspect Window.

You should consider using only strings as the value of TITLE if you are following case 4 above.

4.2.3.2 Inspect Window Properties

PROPERTIES may be interpreted in two ways:

1. If PROPERTIES is a list, it is interpreted as the set of properties of DATUM to be displayed in the Inspect Window. This feature allows you to create Inspect Windows on any data structure in your program, but restrict the set of features that can be examined. Thus, you can provide security for system level properties which you might not want a user to modify.
2. If PROPERTIES is a literal atom, it is applied to DATUM to generate a list of properties which are displayed in the Inspect Window. In effect, the value of PROPERTIES is the name of a function which allows

DISPLAY-ORIENTED TOOLS

you to dynamically determine which properties to display. In addition, you can check whether the user should be allowed to examine selected properties of the object.

Typically, the value of `PROPERTIES` will be the function `GETPROPLIST` which causes the display of all properties associated with the atom specified as the datum.

If you specify no arguments for `INSPECTW.CREATE`, you are prompted for a region for a window which inspects `NIL`. This does not seem to be a particularly useful mechanism.

The properties for the Inspect Window for a text object `{TEXTOBJ}#55,147416`, which is depicted in Figure 4-15, are depicted in Figure 4-16. In this case the value of `PROPERTIES` is the list of record fields to be displayed in the window.

DISPLAY-ORIENTED TOOLS

```

{TEXTOBJ} #55,147416 Inspector
\DIRTY                NIL
PCTB                  {ARRAYP}#64,37130
TEXTLEN               494
\INSERTPC             NIL
\INSERTPCNO           NIL
\INSERTNEXTCH        -1
\INSERTLEFT          0
\INSERTLEN            0
\INSERTSTRING         NIL
\INSERTFIRSTCH       1000000
\INSERTPCVALID       NIL
\WINDOW              NIL
MOUSEREGION          TEXT
LINES                 NIL
DS                    NIL
SEL                   {SELECTION}#55,122630
SCRATCHSEL            {SELECTION}#55,154146
MOVESEL              {SELECTION}#55,122566
SHIFTEDSEL           {SELECTION}#55,122524
DELETEDSEL           {SELECTION}#55,122462
WRIGHT               0
WTOP                 0
WBOTTOM              0
WLEFT                0
TXTFILE               "A complex number is a generalizatio
\XDIRTY              NIL
STREAMHINT            {STREAM}#56,117320
EDITFINISHEDFLG     NIL
CARET                 {EDITCARET}#60,57362
CARETLOOKS           {CHARLOOKS}#61,6240
WINDOWTITLE          NIL
THISLINE             {THISLINE}#61,42644
MENUFLG              NIL
FMTSPEC              {FMTSPEC}#55,127650
FORMATTEDP           NIL
TXTREADONLY          NIL
TXTTERMSA            NIL
EDITOPACTIVE         NIL
DEFAULTCHARLOOKS    {CHARLOOKS}#61,6240
TXTRTBL              NIL
TXTWTBL              NIL
EDITPROPS            (CACHE NIL)
BLUEPENDINGDELETE   NIL
TXTHISTORY           NIL
SELWINDOW            NIL
PROMPTWINDOW         NIL
DISPLAYCACHE         NIL
DISPLAYCACHEDS      NIL
DISPLAYHCPYDS       NIL
TXTPAGEFRAMES       NIL
TYTNEEDSUPDATE      NIL

```

Figure 4-15 Inspect Window for a Text Object

DISPLAY-ORIENTED TOOLS

```
PROPERTIES
(\DIRTY PCTB TEXTLEN \INSERTPC \INSERTPCNO \INSERTNEXTCH \INSERTLEFT \INSERTLEN
 \INSERTSTRING \INSERTFIRSTCH \INSERTPCVALID \WINDOW MOUSEREGION LINES DS SEL
 SCRATCHSEL MOVESEL SHIFTESEL DELETESSEL WRIGHT WTOP WBOTTOM WLEFT TXTFILE
 \XDIRTY STREAMHINT EDITFINISHEDFLG CARET CARETLOOKS WINDOWTITLE THISLINE
 MENUFLG FMTSPEC FORMATTEDP TXTREADONLY TXTTERMSA EDITOPACTIVE DEFAULTCHARLOOKS
 TXTRTBL TXTWTBL EDITPROPS BLUEPENDINGDELETE TXTHISTORY SELWINDOW PROMPTWINDOW
 DISPLAYCACHE DISPLAYCACHEDS DISPLAYHCPYDS TXTPAGEFRAMES TXTNEEDSUPDATE
 TXTCHARLOOKSLIST TXTPARALOOKSLIST)
COPYBUTTONEVENTFN \ITEM.WINDOW.COPY.HANDLER SELECTABLEITEMS
(( (2 594 42 12)
  (DEFAULT.INSPECTW.PROPCOMMANDFN \DIRTY PROPERTY)
  ((129 594 21 12)
   (DEFAULT.INSPECTW.VALUECOMMANDFN NIL (\DIRTY))
```

Figure 4-16 Inspect Window Properties

4.2.3.3 The Fetch Function

The value of **FETCHFN** is a function that returns the value of a property of the object. It takes the form:

Function:	<FETCHFN>
# Arguments:	2
Arguments:	1) OBJECT, an object handle 2) PROPERTY, a property name
Value:	The value of the property, if it exists.

The value is printed in the Inspect Window using PRIN2. For Lisp atoms, the value of <FETCHFN> will usually be the function GETPROP which accesses properties and their values on the atom's property list.

Consider the inspection of the text object {TEXTOBJ}#55,147416. You can find the Inspector properties by inspecting the USERDATA property of the window which appears as a result of executing (INSPECT (WHICHW)) while the cursor resides in the TEXTOBJ Inspect Window. The FETCHFN has the partial definition depicted in Figure 4-17. In this case FETCHFN is

DISPLAY-ORIENTED TOOLS

defined in terms of RECORDACCESS because the text object is defined as a record.

```
FETCHFN
(LAMBDA (INSTANCE FIELD)
 (RECORDACCESS
  FIELD INSTANCE
  (QUOTE
   (DATATYPE TEXTOBJ (** * This is where TEdit stores its state information, and
                        internal data about the text being edited.)
    PCTB
    (* The piece table)
    TEXTLEN
    (* # of chars in the text)
    \INSERTPC
    (* Piece to hold type-in)
    \INSERTPCNO
    (* Piece # of the input piece)
    \INSERTNEXTCH
    (* CH# of next char which is typed into that piece.)
    \INSERTLEFT
    (* Space left in the type-in piece)
    \INSERTLEN
    (* # of characters already in the piece.)
    \INSERTSTRING
    (* The string which the piece describes.)
    \INSERTFIRSTCH
    (* CH# of first char in the piece.)
```

Figure 4-17 FETCHFN Example

4.2.3.4 The Store Function

The value of **STOREFN** is a function that changes the value of a property of the object. It takes the form:

Function:	<STOREFN>
# Arguments:	3
Arguments:	1) OBJECT, an object handle 2) PROPERTY, a property name 3) NEWVALUE, a value expression
Value:	The new value.

STOREFN is invoked by either the default PROPCOMMANDFN or by INSPECTW.REPLACE to change the value of a property. You are encouraged to provide undoable functions as values for this argument.

DISPLAY-ORIENTED TOOLS

For atoms, you should use the function /PUTPROP which is the undoable form of PUTPROP.

The STOREFN for the text object {TEXTOBJ}#55,147416 is depicted in Figure 4-18. It is defined using RECORDACCESS because the text object is defined as a record.

```
(DATUM
{TEXTOBJ}#55,147416 STOREFN
(LAMBDA (INSTANCE FIELD NEWVALUE)
 (RECORDACCESS
  FIELD INSTANCE
  (QUOTE
   (DATATYPE TEXTOBJ (
    (** This is where TEdit stores its state information, and internal data about the text being
    edited.)

    PCTB (* The piece table)
    TEXTLEN (* # of chars in the text)
    \INSERTPC (* Piece to hold type-in)
    \INSERTPCNO (* Piece # of the input piece)
    \INSERTNEXTCH (* CH # of next char which is typed into that piece.)
    \INSERTLEFT (* Space left in the type-in piece)
    \INSERTLEN (* # of characters already in the piece.)
    \INSERTSTRING (* The string which the piece describes.)
    \INSERTFIRSTCH (* CH # of first char in the piece.)
```

Figure 4-18. STOREFN Example

4.2.3.5 The Property Command Function

The value of **PROPCOMMANDFN** is a function which is executed when you press the MIDDLE mouse button while the cursor is pointing at a selected property name in the Inspect Window. It takes the following form:

Function:	<PROPCOMMANDFN>
# Arguments:	3
Arguments:	1) PROPERTY, a property name 2) OBJECT, an object handle 3) INSPECTW, a window handle

DISPLAY-ORIENTED TOOLS

Value: The property name.

The value of **PROPCOMMANDFN** should present the user with a menu of the actions permissible for a property. If you do not specify a value for **PROPCOMMANDFN**, a default function, **DEFAULT.INSPECTW.PROPCOMMANDFN**, will be assigned as its value when the Inspect Window is created. This function presents a menu of one item, **SET**.

If you select **SET**, it reads a value that you type in, evaluates it, and assigns it as the value of the property. A sample entry for **\DIRTY** appears in the **SELECTABLEITEMS** field. The **PROPCOMMANDFN** for the **\DIRTY** property of the text object **{TEXTOBJ}#55,147416** appears as:

```
((2 594 42 12)
DEFAULT.INSPECTW.PROPCOMMANDFN \DIRTY
PROPERTY)
```

The first element of the list is the region specification in the Inspect Window for the property.

If you wish to disable the **PROPCOMMANDFN**, you may assign a string as its value which will be displayed in the Prompt Window when a property is selected.

4.2.3.6 The Value Command Function

The value of **VALUECOMMANDFN** is a function that is executed when you press the **MIDDLE** mouse button while the cursor is pointing to a selected property value in the Inspect Window. It takes the form:

Function: <VALUECOMMANDFN>
Arguments: 4

DISPLAY-ORIENTED TOOLS

Arguments: 1) VALUE, the selected value
 2) PROPERTY, the property name
 3) OBJECT, an object handle
 4) INSPECTW, a window handle
Value: The new value.

In executing the function, the Inspector calls the value of `FETCHFN` to retrieve the value of the property of the object. If the value of `VALUECOMMANDFN` is `NIL`, a default function, `DEFAULT.INSPECT.VALUECOMMANDFN`, is assigned as its value when the Inspect Window is created. It presents a menu of the possible ways to inspect the value (see Section 4.2.4, II). When one of these methods is selected, it creates a new Inspect Window to perform it.

4.2.3.7 The Title Command Function

The value of `TITLECOMMANDFN` is a function which is executed when you press the `MIDDLE` mouse button while the cursor is situated in the title or border area of the Inspect Window. It takes the form:

Function: <TITLECOMMANDFN>
Arguments: 2
Arguments: 1) INSPECTW, a window handle
 2) OBJECT, an object handle
Value: The window handle.

The primary use of `TITLECOMMANDFN` is to allow you to select functions which might be applied to the entire object rather than a single property. If the value of `TITLECOMMANDFN` is `NIL`, a default function, `DEFAULT.INSPECTW.TITLECOMMANDFN`, is assigned as its value when the Inspect Window is created. This function presents

a menu of standard operations as described in Section 4.2.4 of *Medley-Interlisp:Interactive Programming Environment*.

4.2.3.8 The Selection Function

The value of **SELECTIONFN** is a function which is executed when you release the LEFT mouse button while the cursor is pointing at one of the items in the Inspect Window. It takes the form:

Function:	<SELECTIONFN>
# Arguments:	3
Arguments:	1) PROPERTY, a property name 2) VALUEFLG, a flag for the item type 3) INSPECTW, a window handle
Value:	The property

This function allows you to take action with respect to one of the items displayed in the Inspect Window. When it is called, the selected item is highlighted to indicate that it is "selected".

4.2.3.9 The Property Printing Function

The value of **PROPPRINTFN** is a function which is executed to determine what to print in the property place for the given property. It takes the form:

Function:	<PROPPRINTFN>
# Arguments:	2
Arguments:	1) PROPERTY, a property name 2) DATUM, the datum handle
Value:	The entry to be printed

This function allows you to substitute some other string or atom for the specified property name when the property is to be displayed

in the Inspect Window. If its value is NIL, no property name will be printed.

4.2.3.10 Redisplaying an Inspect Window

You may redisplay the object that is viewed by an Inspect Window using the function **INSPECTW.REDISPLAY**, which takes the form:

Function:	INSPECTW.REDISPLAY
# Arguments:	2
Arguments:	1) INSPECTW, a window handle 2) PROPS, a list of properties
Value:	The window handle.

INSPECTW.REDISPLAY updates the Inspect Window which is displaying one or more objects. Inspect windows do not automatically update their display when the object they are showing is updated. Thus, if you are using Inspect Windows to monitor the changing values of objects within a system (such as for debugging or process monitoring), every time that a property is updated, you must force a redisplay of the associated Inspect Window.

PROPS is generally interpreted as a list of properties to be updated within an Inspect Window. If PROPS is NIL, all properties will be updated when this function is executed.

This function is called by the *ReFetch* command in the title command menu of an Inspect Window.

4.2.3.11 Replacing Property Values

The function **INSPECTW.REPLACE** is provided as a functional interface for implementing your PROPCOMMANDFN. It takes the form:

DISPLAY-ORIENTED TOOLS

Function:	INSPECTW.REPLACE
# Arguments:	3
Arguments:	1) INSPECTW, a window handle 2) PROPERTY, a property name 3) NEWVALUE, an expression
Value:	The region of the value of the property.

INSPECTW.REPLACE invokes the STOREFN to change the value of the property and calls INSPECTW.REDISPLAY to update the display in the Inspect Window. Consider the following example (depicted in Figure 4-19):

```
<-(INSPECTW.REPLACE IW 'WTOP 20)  
(129 342 14 12)
```

DISPLAY-ORIENTED TOOLS

```
Enter the new WINDOWTITLE for {TEXTOBJ}#55,147416
The expression read will be EVALuated.
> "MYTEXT WINDOW"

{TEXTOBJ}#55,147416 Inspector
\DIRTY                NIL
PCTB                  {ARRAYP}#64,37130
TEXTLEN               494
\INSERTPC             NIL
\INSERTPCNO           NIL
\INSERTNEXTCH        -1
\INSERTLEFT           0
\INSERTLEN            0
\INSERTSTRING         NIL
\INSERTFIRSTCH       1000000
\INSERTPCVALID       NIL
\WINDOW               NIL
MOUSEREGION          TEXT
LINES                 NIL
DS                    NIL
SEL                   {SELECTION}#55,122630
SCRATCHSEL            {SELECTION}#55,154146
MOVESEL               {SELECTION}#55,122566
SHIFTEDSEL            {SELECTION}#55,122524
DELETSEL              {SELECTION}#55,122462
WRIGHT                0
WTOP                  20
WBOTTOM               0
WLEFT                 0
```

Figure 4-19 INSPECTW.REPLACE Example

4.2.3.12 Selecting an Item in an Inspect Window

When you point the cursor at an item in an Inspect Window and press the LEFT mouse button, the item is selected. Pressing the LEFT mouse button invokes the function **INSPECTW.SELECTITEM**, which takes the form:

Function:	INSPECTW.SELECTITEM
# Arguments:	3
Arguments:	1) INSPECTW, a window handle 2) PROPERTY, a property name 3) VALUEFLG, a flag for the item

DISPLAY-ORIENTED TOOLS

Value: type
 NIL.

When an item is selected, `INSPECTW.SELECTITEM` inverts the item's name in the Inspect Window and assigns it as the value of the window property `CURRENTITEM`. If `CURRENTITEM` was non-NIL, it is deselected.

`VALUEFLG` determines whether the item selected was the property name (value is NIL) or the property value (value is T).

If `PROPERTY` is NIL, no item will be selected. This feature allows you to select all items in the Inspect Window.

Consider the following example (see Figure 4-20):

```
<-(INSPECTW.SELECTITEM IW 'WLEFT)  
NIL
```

```
<-(WINDOWPROP IW 'CURRENTITEM)  
((2 318 35 12) DEFAULT.INSPECTW.PROPCOMMANDFN  
WLEFT PROPERTY)
```

DISPLAY-ORIENTED TOOLS

```
{TEXTOBJ} # 55,147416 Inspector
\DIRTY          NIL
PCTB            {ARRAYP}#64,37130
TEXTLEN        494
\INSERTPC      NIL
\INSERTPCNO    NIL
\INSERTNEXTCH  -1
\INSERTLEFT    0
\INSERTLEN     0
\INSERTSTRING  NIL
\INSERTFIRSTCH 1000000
\INSERTPCVALID NIL
\WINDOW        NIL
MOUSEREGION    TEXT
LINES          NIL
DS             NIL
SEL            {SELECTION}#55,122630
SCRATCHSEL     {SELECTION}#55,154146
MOVESEL        {SELECTION}#55,122566
SHIFTEDSEL     {SELECTION}#55,122524
DELETEDSEL     {SELECTION}#55,122462
WRIGHT         0
WTOP           20
WBOTTOM        0
WLEFT          0
TXTFILE        "A complex number is a generalizatio
\XDIRTY        NIL
STREAMHINT     {STREAM}#56,117320
EDITFINISHEDFLG NIL
CARET          {TEDITCARET}#60,57362
CARETLOOKS    {CHARLOOKS}#61,6240
```

Figure 4-20 INSPECTW.SELECTITEM Example

4.2.3.11 Obtaining the INSPECTW Properties

You can retrieve the properties that are displayed in an Inspect Window using the function **INSPECTW.PROPERTIES**, which takes the form:

Function:	INSPECTW.PROPERTIES
# Arguments:	1
Arguments:	1) INSPECTW, the window handle

DISPLAY-ORIENTED TOOLS

Value: A list of the properties.

When you execute this function, you are prompted by a menu to select the method of display: DisplayEdit, Inspect, As a Record, or As a PLIST, I selected DisplayEdit for the following example:

```
<-IW
{WINDOW}#55,120300

<-(INSPECTW.PROPERTIES IW)
(  \DIRTY
   PCTB
   TEXTLEN
   \INSERTPC
   |INSERTPCNO
   \INSERTNEXTCH
   \INSERTLEFT
   \INSERTLEN
   \INSERTSTRING
   \INSERTFIRSTCH
   \INSERTPCVALID
   \WINDOW
   MOUSEREGION
   LINES
   DS
   SEL
   SCRATCHSEL
   MOVESEL
   SHIFTESEL
   DELETESEL ...
   and so forth
)
```

4.2.3.12 Fetching the Value of a Property

DISPLAY-ORIENTED TOOLS

You may fetch the value of a property displayed in an Inspect Window using the function **INSPECTW.FETCH**, which takes the form:

Function:	INSPECTW.FETCH
# Arguments:	2
Arguments:	1) INSPECTW, the window handle 2) PROPERTY, the name of a property
Value:	The property's value.

PROPERTY must be one of the properties in the list of properties specified when you created the Inspect Window. Consider the following example:

```
<-(INSPECTW.FETCH IW 'CARETLOOKS)
{CHARLOOKS}#61,6240
```

4.2.4 Inspect Commands

The middle mouse button invokes a pop-up menu of commands when an item is selected within an inspect window. The type of menu depends on whether the item selected was a property name or a value.

4.2.4.1 Property Name Commands

When you select a property name in an Inspect Window, it is highlighted by *reverse video*. Pressing the middle mouse button causes a pop-up menu to appear with commands that can operate on the property. The standard menu includes a single command SET which allows you to change the property's value. Figure 4-19 depicts the prompt pane that appears. Evaluation is initiated by pressing a carriage return or completing an S-expression.

4.2.4.2 Value Commands

When you select a value, it is highlighted by reverse video. Pressing the middle mouse button causes a pop-up menu to appear with commands that can operate on the value. The commands that appear depend on the type of value. For example, the value for `DEFAULTCHARLOOKS` is a `CHARLOOKS` object. The command that appears for it is `INSPECT`, whereas the commands that appear for the value of `EDITPROPS` are `DisplayEdit`, `TtyEdit`, `As a Record`, and `As a PLIST`.

4.2.4.3 Inspect Window Commands

If you place the cursor on the title bar of the Inspector Window and press the middle mouse button, a menu of commands that applies to the Inspector Window itself will pop-up. These commands are presented in Table 4-3.

Table 4-3. Inspect Window Middle Mouse Button Commands

Command	Description
ReFetch	Updates the values of the object which is displayed in the Inspector Window if it has changed since the last time those values were fetched
IT<-datum	Sets the variable IT to the object being inspected in the Inspector Window.
IT<-selection	Sets the variable IT to the property name or value which is currently selected within the Inspector Window.

4.2.5 Interacting with Break Windows

The Break Package in the Interlisp environment is integrated with the Inspector. Thus, when you are examining a back trace of the stack in a Break Window, you may inspect the the contents of a

particular frame in the stack. To do so, you select the frame name in the Break Window stack frame menu and press the left mouse button.

When you examine a frame, which is displayed in a separate window, you may access the named objects and their values. Of course, some objects will not have inspectable features and you will be informed by a message in the Prompt Window. You may change the values of objects bound in a frame by:

1. Selecting the object name in the Inspect Window.
2. Pressing the MIDDLE button to pop-up a menu of commands.
3. Selecting SET from the menu.

Caution should be exercised when changing the values of variables on the stack. Although Interlisp allows you to inspect the compiled versions of system functions, you should not attempt to change the values of objects bound within system function frames. Setting system variables to unusual values can crash the system.

4.2.6 Inspector Variables

The behavior of the Inspector is controlled by a number of system variables. Generally, these control the amount of information to be displayed when you invoke the Inspector.

4.2.6.1 Handling Long Lists

It is possible, perhaps even normal, to create very long lists in Interlisp programs. When inspecting the value of a list, you may not know how long it is. As mentioned above, the Inspector sizes the window in which it displays values of objects and makes them scrollable if the length of the value of the object exceeds the window size.

Waiting for the Inspector to display the entire value of a list can be tedious, especially when it has to set up a scrolling option. Often, you can determine from the first N elements of a list what the problem.

You can set the number of elements of a list that the Inspector will display by setting the system variable `MAXINSPECTCDRLEVEL`. Initially, its value is 50.

When a long list is inspected, the last item to be displayed is set to contain the unprinted elements. You may inspect the tail to see the remaining elements (or at least the next elements limited by `MAXINSPECTCDRLEVEL`).

4.2.6.2 Inspecting Arrays

You will often face a similar problem when inspecting arrays. You can set the number of elements of an array that the Inspector will display by setting the value of `MAXINSPECTARRAYLEVEL`. Initially, its value is 300. You may inspect the remaining elements by calling the function `INSPECT/ARRAY`.

4.2.6.3 Setting `PRINTLEVEL`

When printing values in an Inspect Window, the Inspector uses the standard system printing functions. The behavior of these printing functions is mediated by the value of `PRINTLEVEL`. The Inspector temporarily sets the value of `PRINTLEVEL` to the value of `INSPECTPRINTLEVEL` while printing values in an Inspect Window. Its value is initially (2 . 5).

4.2.6.4 Inspecting Record Fields

When you inspect the values of objects which have record definitions, the Inspector uses the value of

DISPLAY-ORIENTED TOOLS

INSPECTALLFIELDSFLG to determine which fields of the definition to display. Initially, its value is T. Thus, the Inspector will display both computed fields (using ACCESSFNS) and regular fields of the record definition.

4.2.7 Inspector Macros

The Inspector "knows" about the standard data types and objects that are defined within Interlisp. When you create new data objects, you can tell the Inspector about them by defining Inspector Macros. Inspector Macros are defined as entries on the list *INSPECTMACROS*. Each entry takes the form:

(<OBJECTTYPE> . <INSPECTINFO>)

where:

OBJECTTYPE	Specifies the types of objects that may be inspected with this macro.
INSPECTINFO	Specifies the definition of the macro.

The initial value of INSPECTMACROS is:

```
((READTABLEP      RDTBL\NONOTHERCODES
                   GETSYNTAXPROP
                   SETSYNTAXPROP)
 (TERMTABLEP      (CHARDELETE
                   WORDELETE
                   LINEDELTE
                   RETYPE
                   CTRLV
                   EOL
                   RAISE
                   ECHOMODE
                   LINEDELETISTR
                   1STCHDEL
```

DISPLAY-ORIENTED TOOLS

```
NTHCHDEL
POSTCHDEL
EMPTYCHDEL
ECHODELS?
CONTROL
0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31)
GETTTBLPROP
SETTBLPROP
))
```

4.2.7.1 Macros for Literal Atoms

If OBJECTTYPE is a literal atom (i.e., satisfies LITATOM), the value of INSPECTINFO will be used to inspect items whose type name is OBJECTTYPE.

4.2.7.2 Macros for Lists

If OBJECTTYPE is a list of the form:

```
(<function> <datum-predicate>)
```

DATUM-PREDICATE will be applied to the item. If it returns a non-NIL value, then the INSPECTINFO value will be used to inspect the item. INSPECTINFO can take one of two forms:

1. If INSPECTINFO is a literal atom (i.e., satisfies LITATOM), it should be a function of three arguments: the item being inspected, the value of OBJECTTYPE, and the value of the WHERE argument of INSPECT. The function will do the inspecting.
2. If INSPECTINFO is not a literal atom, it should be a list of the form:

```
((<properties>
  <fetchfn>
  <storefn>
  <propcommandfn>
  <valuecommandfn>
  <titlecommandfn>
  <title>
  <selectionfn>
  <where>
  <propprintfn>)
```

which are just the arguments that would be passed to `INSPECTW.CREATE`. The `WHERE` argument is evaluated, but the others are not.

4.3 The Grapher Utility

Although Grapher is a Lisp Library Package, its usefulness is so ubiquitous that I describe it with the other major interactive display tools.

Grapher is a package of functions that allows you to layout, display, and edit graphs configured as networks of nodes and links. Each node in a graph has a label, but the links do not. Links are drawn as straight lines between nodes by default. You may adjust the appearance of individual links. Node labels can be text, bitmaps, or image objects. You can attach functions and menus to individual nodes such that the functions are triggered or the menus are displayed when a node is selected with the mouse.

The following data structure, which is assigned to the variable `PARSE`, will be used to demonstrate the functions associated with Grapher:

DISPLAY-ORIENTED TOOLS

```
(SENTENCE
  ((TENSE PAST)
  (SENTENCE-TYPE DECLARATIVE)
    (NUMPERS (1 3)))
  (SUBJECT
    ((NUMPERS (1 3)))
  (NOUN-PHRASE-HEAD
    ((NUMPERS (1 3)))
    HE))
  (PREDICATE
    ((COMPLEMENT NIL)
    (TENSE PAST)
    (VNUMPERS (1 3))
    (TRANSITIVE T))
    GAVE
  (OBJECT
    ((NUMPERS (1 3)))
  (NOUN-PHRASE-HEAD
    ((NUMPERS (1 3)))
    THE BOOK))
)
```

which is a syntactic parse of the sentence "He gave the book".

4.3.1 The Structure of a Graph

A *graph* is represented by a GRAPH record which has the described in Table 4-4.

Table 4-4. Graph Record Fields

Field	Usage
GRAPHNODES	A list of graph nodes which have the structure described below.

DISPLAY-ORIENTED TOOLS

DIRECTEDFLG	Controls how links are drawn between nodes.
SIDESFLG	Controls the drawing of links between the sides of nodes rather than their top or bottom edges.
GRAPH.MOVENODEFN	If non-NIL, a function that is called after you have moved a node in the graph.
GRAPH.ADDNODEFN	If non-NIL, a function that is called to add a node to the graph.
GRAPH.DELETENODEFN	If non-NIL, a function that is called when a node is to be deleted from the graph.
GRAPH.ADDLINKFN	If non-NIL, a function that is called when a link is added between two nodes.
GRAPH.DELETELINKFN	If non-NIL, a function that is called when you are about to delete a link between two nodes.
GRAPH.FONTCHANGEFN	If non-NIL, a function that allows you to change the size of node labels.

4.3.1.1 DIRECTEDFLG

If `DIRECTEDFLG` is `NIL`, Grapher will draw each link so that it does not cross the node labels of the nodes that it runs between. The objective is not to obscure the node labels by "running" over them with the link lines. However, this may cause ambiguities in the visual interpretation of the graph which are settled by `SIDESFLG`.

If `DIRECTFLG` is non-`NIL`, the edges are always directed to the left edge of the `To` node. But, this may cause links to cross the labels of the nodes that they run between. `SIDESFLG` is used to adjust the position of the link.

4.3.1.2 SIDESFLG

DISPLAY-ORIENTED TOOLS

SIDESFLG is used to determine which side of a node the links will be drawn to or from. When DIRECTEDFLG is NIL, SIDESFLG is interpreted as presented in Table 4-5.

Table 4-5. SIDESFLG Interpretation

Value	Usage
NIL	The links will be drawn between the top and bottom edges of the nodes.
non-NIL	The links are drawn between the sides of the nodes.

When DIRECTEDFLG is non-NIL, SIDESFLG is interpreted as as presented in Table 4-6.

Table 4-6. SIDESFLG Interpretation

Value	Usage
NIL	The From end of the link is drawn to the bottom edge of the From node and the To end of the link is drawn to the top edge of the To node.
non-NIL	The From end of the link is drawn to the right edge of the From node and the To end of the link is drawn to the left edge of the To node.

4.3.1.3 GRAPH.MOVENODEFN

The value of the GRAPH.MOVENODEFN field is a function which takes the form:

Function: <graph.movenodefn>
Arguments: 5
Arguments: 1) NODE, a node identifier
2) NEWPOSITION, a new position for the node

DISPLAY-ORIENTED TOOLS

- 3) GRAPH, a graph structure
 - 4) WINDOW/STREAM, a window or display stream handle
 - 5) OLDPOSITION, the old position of the node
- Value: <user-defined>

After you move a node interactively, this function is called with the arguments indicated. This function performs any auxiliary tasks that you may desire, including moving other nodes a like distance. The distance can be computed from the difference of NEWPOSITION and OLDPOSITION.

4.3.1.4 GRAPH.ADDNODEFN

The value of the GRAPH.ADDNODEFN field is a function which takes the form:

Function: <graph.addnodefn>
#Arguments: 2
Arguments: 1) GRAPH, a graph structure
2) WINDOW, a window handle
Value: <user defined>

When you select the "Add a node" item from the graph editing menu, this function will be invoked. It should return a node which is added to the graph structure or NIL if no node is to be added to the graph. After the node is added, it is positioned within the graph.

4.3.1.4 GRAPH.DELETENODEFN

DISPLAY-ORIENTED TOOLS

The value of the GRAPH.DELETENODEFN field is a function which takes the form:

Function:	<graph.deletenodefn>
# Arguments:	3
Arguments:	1) NODE, a graphnode record 2) GRAPH, a graph record 3) WINDOW, a window handle
Value:	<user defined>

After you delete a node interactively, this function is called with the arguments indicated. This function performs any auxiliary tasks that you may desire, including deleting other nodes (such as dangling nodes after an intermediate node is deleted) or repositioning nodes in the graph.

4.3.1.5 GRAPH.ADDLINKFN

The value of the GRAPH.ADDLINKFN field is a function which takes the form:

Function:	<graph.addlinkfn>
# Arguments:	4
Arguments:	1) FROM, the from node of the link 2) TO, the to node of the link 3) GRAPH, a graph record 4) WINDOW, a window handle
Value:	<user defined>

After the link between the nodes FROM and TO has been added interactively, this function is called with the arguments indicated.

DISPLAY-ORIENTED TOOLS

This function performs any auxiliary tasks that you may desire, including repositioning nodes in the graph.

4.3.1.6 GRAPH.DELETELINKFN

The value of the **GRAPH.DELETELINKFN** field is a function which takes the form:

Function:	<graph.deletelinkfn>
# Arguments:	4
Arguments:	1) FROM, the from node of the link 2) TO, the to node of the link 3) GRAPH, a graph record 4) WINDOW, a window handle
Value:	<user defined>

After the link between the nodes FROM and TO has been deleted interactively, this function is called with the arguments indicated. This function performs any auxiliary tasks that you may desire, including repositioning nodes in the graph or possibly deleting a dangling node. Note that this function may also be called after you delete a node whence any dangling links are also deleted.

4.3.1.7 GRAPH.FONTCHANGEFN

The value of the **GRAPH.FONTCHANGEFN** field is a function which takes the form:

Function:	<graph.fontchangefn>
# Arguments:	4
Arguments:	1) HOW, one of LARGER or SMALLER 2) NODE, a graphnode record

DISPLAY-ORIENTED TOOLS

- Value:
- 3) GRAPH, a graph record
 - 4) WINDOW, a window handle
- <user defined>

This function is called after you have selected either of the commands "label smaller" or "label larger" from the EDITGRAPHMENU.

4.3.2 The Structure of a Graph Node

Each graph node is represented by a GRAPHNODE record which has the fields presented in Table 4-7.

Table 4-7. GRAPHNODE Fields

Field	Usage
NODELABEL	The label displayed in the node.
NODEID	A unique identifier for the node.
TONODES	A list of all nodes to which links run from this node.
FROMNODES	A list of all nodes from which links run to this node.
NODEPOSITION	The location of the center of the node
NODEFONT	The font in which the node's label will be displayed.
NODEBORDER	The shade and width of the border around the node.
NODELABELSHADE	The background shade of the node.
NODEWIDTH	The width of the node's NODELABEL.
NODEHEIGHT	The height of the node's NODELABEL.

These fields are described in more detail in the following sections.

4.3.2.1 The Node Label

The *node label* is the name of the node which you see when the node is displayed in the graph. The node label may be

1. a string
2. a bitmap
3. an image object

If the node label is a text string, it is printed in the node via PRIN3.

If the node is a bitmap, it is bit-blitted into the node where the size of the node is used as the clipping region.

If the node is an image object, then its IMAGEBOXFN and DISPLAYFN are used to display the object.

All other Lisp objects are printed using PRIN3. Thus, the node label could be the handle of an array, the name of an atom, or a number.

4.3.2.2 The Node Identifier

The *node identifier* is a unique identifier for a node. It is used in the link field rather than a pointer to the node itself so that circular Lisp structures can be avoided. A node identifier may be used as the pointer to the structure represented by the graph.

4.3.2.3 Links To Other Nodes

The *TONODES* field is a list of the nodes to which a link runs from this node. Each entry may be a link description as discussed in Section 4.3.3.

4.3.2.4 Links From Other Nodes

The *FROMNODES* field is a list of nodes from which a link runs to this node.

4.3.2.5 The Node Position

The *node position* specifies the position of the center of the node in the coordinates of the display stream in which the node is to be displayed.

4.3.2.6 The Node Font

The *node font* is the font in which the node's label will be displayed if it is a text string. Its value may be any font specification that can be used by FONTCREATE to generate a font descriptor.

If it is NIL, the default font is found in the system variable DEFAULT.GRAPH.NODEFONT, which is usually NIL to indicate the system default font.

4.3.2.7 The Border of the Node

The *node border* specifies the shade and width of the border surrounding the node. It may take the values presented in Table 4-8.

Table 4-8. Graphnode Border Values

Value	Usage
NIL	No border.
0	A border of zero width (e.g., no border).
T	Black border which is one pixel wide.
1,2,3,...	Black border of a width of the number of pixels.
-1,-2,...	White border of the width of the number of pixels.

DISPLAY-ORIENTED TOOLS

(W S)	W is an integer and S is a texture handle or shade specification; displays a border W pixels wide filled with the given shade or texture.
-------	---

4.3.2.8 The Shade of the Node Label

The *node label shade* specifies the background shade used to fill the node in order to highlight the node label. However, this only works for node labels which are text strings or arbitrary Lisp objects other than bitmaps and image objects. It may take the values presented in Table 4-9.

Table 4-9. GraphNode Shade Values

Value	Usage
NIL	Equivalent to WhiteShade
T	Equivalent to BLACKSHADE
a <texture>	A texture object
a <bitmap>	A bitmap object

The default value is found in the system variable `DEFAULT.GRAPH.NODELABELSHADE`, which is initially `NIL`.

4.3.2.9 The Node Width and Height

The *node width* and *node height* specify the width and height of the node. Initially, these are set to be the width and height of the node's label, respectively.

4.3.3 The Structure of a Link Description

Entries for the `TONODES` field can describe the appearance of the link to be drawn between the source node and the target node specified in the link description. The *link description* entry is a list having the following form:

(Link%Parameters <ToNodeId> . <ParamList>)

which is interpreted as follows:

- Link%Parameters: a keyword identifying the list as having a special meaning.
- ToNodeId: is the To node from which a link will be drawn from the current node.
- ParamList: is a list of parameters affecting the appearance of the link.

The properties which may be specified in ParamList are found as the value of the variable ScalableLinkParameters. Its initial value is:

<-ScalableLinkParameters
(DASHING LINEWIDTH)

4.3.4 Creating a Node

You may create a GRAPHNODE record for a new graph node using the function **NODECREATE**, which takes the form:

Function:	NODECREATE
# Arguments:	8
Arguments:	1) ID, the node identifier
	2) LABEL, the node label
	3) POSITION, the node position
	4) TONODEIDS, a list of nodes to which links will be drawn from this node
	5) FROMNODEIDS, a list of nodes which links will be drawn to this node
	6) FONT, the node font

DISPLAY-ORIENTED TOOLS

- 7) BORDER, the node border
- 8) LABELSHADE, the node label shade

Value: A GRAPHNODE record.

NODECREATE returns a GRAPHNODE which has NODELABEL = ID, NODEPOSITION = POSITION, NODEFONT = FONT, and BOXNODEFLG = BOXED?. The set of nodes should be doubly linked so that if a node A is on the TONODES of B, then B is on the FROMNODES of A. You must make sure that this constraint is satisfied or LAYOUTGRAPH may not be able to completely layout the graph.

The arguments given to NODECREATE have the obvious interpretations corresponding to the fields of the GRAPHNODE record as described in Section 4.3.2. Consider the following examples:

```
<-(SETQ SNODE (NODECREATE 1 'SENTENCE NIL '(2 3 4))
(1 T NIL NIL NIL NIL (2 3 4)
  NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  SENTENCE NIL)
```

```
<-(SETQ SUNODE (NODECREATE 2 'SUBJECT '(1) '(5))
(2 T NIL NIL NIL NIL (5)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  SUBJECT NIL)
```

```
<-(SETQ PNODE (NODECREATE 3 'PREDICATE '(1) '(9))
(3 T NIL NIL NIL NIL (9)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  PREDICATE NIL)
```


DISPLAY-ORIENTED TOOLS

```
<-(SETQ ONODE (NODECREATE 4 'OBJECT '(1) '(6))
(4 T NIL NIL NIL NIL NIL (6)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  OBJECT NIL)
```

```
<-(SETQ NPHNODE-1 (NODECREATE 5 'NOUN-PHRASE-
HEAD '(2) '(7))
(5 T NIL NIL NIL NIL NIL (7)
  (2)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  NOUN-PHRASE-HEAD NIL)
```

```
<-(SETQ NPHNODE-2 (NODECREATE 6 'NOUN-PHRASE-
HEAD '(4) '(8))
(6 T NIL NIL NIL NIL NIL (8)
  (4)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  NOUN-PHRASE-HEAD NIL)
```

```
<-(SETQ HENODE (NODECREATE 7 'HE '(5) NIL))
(7 NIL NIL NIL NIL NIL NIL NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  HE NIL)
```

```
<-(SETQ PARKNODE (NODECREATE 7 '(THE PARK) '(6)
NIL))
(8 NIL NIL NIL NIL NIL NIL NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  (THE PARK) NIL)
```

```
<-(SETQ GAVENODE (NODECREATE 9 'HE '(3) NIL))
(9 NIL NIL NIL NIL NIL NIL NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
```

DISPLAY-ORIENTED TOOLS

GAVE NIL)

At a minimum, you must specify the node identifier, the label of the node, the *identifiers* of parent nodes, and the *identifiers*, if any, of descendent nodes. Other arguments are optional and defaults are assumed.

Now, we can create a list of graph nodes for use by LAYOUTGRAPH as follows:

```
<-(SETQ GRAPHNODELST
  (LIST SNODE SUNODE PNODE ONODE NPHNODE-1
    NPHNODE-2
      HENODE GAVENODE PARKNODE))
((1 T NIL NIL NIL NIL NIL (2 3 4)
  NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  SENTENCE NIL)
 (2 T NIL NIL NIL NIL NIL (5)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  SUBJECT NIL)
 (3 T NIL NIL NIL NIL NIL (9)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  PREDICATE NIL)
 (4 T NIL NIL NIL NIL NIL (6)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  OBJECT NIL)
 (5 T NIL NIL NIL NIL NIL (7)
  (2)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  NOUN-PHRASE-HEAD NIL)
 (6 T NIL NIL NIL NIL NIL (8)
```

DISPLAY-ORIENTED TOOLS

- (4)
(GACHA 10 (MEDIUM REGULAR REGULAR))
NOUN-PHRASE-HEAD NIL)
- (7 NIL NIL NIL NIL NIL NIL NIL NIL
(GACHA 10 (MEDIUM REGULAR REGULAR))
HE NIL)
- (8 NIL NIL NIL NIL NIL NIL NIL NIL
(GACHA 10 (MEDIUM REGULAR REGULAR))
(THE PARK) NIL)
- (9 NIL NIL NIL NIL NIL NIL NIL NIL
(GACHA 10 (MEDIUM REGULAR REGULAR))
GAVE NIL))

4.3.5 Displaying a Graph

You may display a graph on the screen using the function `SHOWGRAPH`, which takes the form:

Function:	<code>SHOWGRAPH</code>
# Arguments:	7
Arguments:	1) <code>GRAPH</code> , a list of the nodes of the graph 2) <code>WINDOW</code> , the window in which the graph will be displayed 3) <code>LEFTBUTTONFN</code> , a function for the left mouse button 4) <code>MIDDLEBUTTONFN</code> , a function for the middle mouse button 5) <code>TOPJUSTIFYFLG</code> , a flag determining the initial position of the graph 6) <code>ALLOWEDITFLG</code> , enables editing via the right mouse button 7) <code>COPYBUTTONEVENTFN</code> , a function for handling

DISPLAY-ORIENTED TOOLS

Value: copy-select operations
The window handle.

SHOWGRAPH displays the nodes of the graph in the specified window, if it exists. If WINDOW is NIL, the graph is displayed in a window which is sized to the graph. Consider the following example:

```
<-(SHOWGRAPH vertgraph)
{WINDOW}#74,25000
```

which is depicted in Figure 4-21. The graph is displayed in a vertical orientation because the VERTICAL option was specified when the graph was laid out by LAYOUTSEXPR.

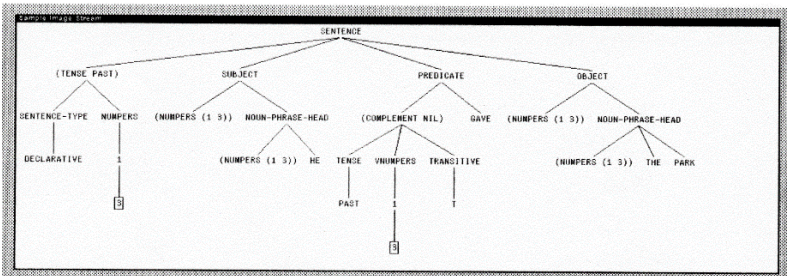


Figure 4-21 Example of SHOWGRAPH

If WINDOW is a string, the graph is displayed in a window that is large enough to hold it and WINDOW becomes the title of the window.

If WINDOW exists and is closed, GRAPHER opens the window to display the graph. If the graph is larger than the window, the window is made scrollable and the leftmost, topmost portion of the graph is initially displayed.

4.3.5.1 Justifying the Graph

TOPJUSTIFYFLG determines where the graph is initially positioned within the window. If TOPJUSTIFYFLG is T, the graph's top edge will be displayed at the top edge of the window. If TOPJUSTIFYFLG is NIL, the graph's bottom edge is displayed at the bottom edge of the window.

You may reshape the window containing the graph using the normal shaping commands in the standard window menu. The list representing the graph is stored as the value of the GRAPH property on the window handle.

4.3.5.2 Button Event Functions

If either LEFTBUTTONFN or MIDDLEBUTTONFN or both are non-NIL, the window is provided with a BUTTONEVENTFN which turns the graph into a menu. Whenever the cursor is positioned over a node of the graph and you press the left or middle button, the node will be inverted and the appropriate button function will be called.

These functions are called with two arguments: the selected node (e.g., a GRAPHNODE record) and the window. The button function may access the entire graph through the window's GRAPH property.

4.3.5.3 Editing a Graph

If the argument ALLOWEDITFLG is non-NIL, the right mouse button allows you to select an editing function from a menu displayed when you press the button while the mouse is positioned in the Grapher window. The functions in the EDITGRAPHMENU are:

- Move Node

DISPLAY-ORIENTED TOOLS

- Add Node
- Delete Node
- Add Link
- Delete Link
- Change Label
- label smaller
- label larger
- <-> Directed
- <-> Sides
- <-> Border
- <-> Shade

These operations are described in Table 4-10.

Table 4-10. Graph Editing Functions

Function	Description
Moving a Node	The <i>Move Node</i> operation allows you to move a node of the graph. You are prompted to select a node with the cursor. You press the left mouse button and "drag" the node to its new position in the graph. Note that the links connecting the selected node to other nodes follow the node as it moves. GRAPHER automatically adjusts the length of the links as you drag the node.
Add a Node	The <i>Add Node</i> operation allows you to create a new node for the graph. You are prompted by a pop-up window to enter the label for the new node. You terminate the new node label with a <CR>. The new node is automatically drawn with a border. You can position the node by pressing the left mouse button while the cursor rests on the node and dragging it to its new position. Figure 4-22 depicts the graph with four new nodes added (around the area of INDIRECT OBJECT).

DISPLAY-ORIENTED TOOLS

Delete a Node	The <i>Delete Node</i> operation allows you to delete a node from a graph. You are prompted to select a node with the cursor. It is highlighted in reverse video and you are prompted for confirmation. Answering "y" causes the node to be deleted and any links it has to other nodes. Note that you may delete intermediate nodes in the graph which will leave other nodes dangling.
Change Label	The <i>Change Label</i> operation allows you to change the name of a node. After selecting the node, you are prompted, via a pop-up window, to enter the new node label. You terminate the new node label by <CR>.
Adding a Link	The <i>Add Link</i> operation allows you to add a new link between two nodes. You are prompted to select the FROM and TO nodes with the mouse. A link is drawn between the two nodes. Figure 4-23 depicts the graph with links connecting the INDIRECT OBJECT nodes.
Deleting a Link	The <i>Delete Link</i> operation allows you to delete a link between two nodes. You are prompted to select the nodes with the mouse. Note that you may leave a dangling node by deleting a link between it and another node.
Making a Label Smaller	The <i>label smaller</i> operation allows you to reduce the font size of a node label. GRAPHER reduces the label to the next smallest font size that it knows about for a given font. You may continue this operation until you exhaust the available font files.
Making a Label Larger	The <i>label larger</i> operation allows you to incrementally increase the size of a node label. GRAPHER increases the node label to the next largest font size that it knows about for the given font. You may continue to increase the label size until you exhaust the available font files. Figure 4-24 depicts the graph after several applications of these two commands.

DISPLAY-ORIENTED TOOLS

<-> Sides	The <-> <i>Sides</i> operation moves the links from the left or right sides of a node to the top or bottom of a node. It is mainly used for cosmetic adjustments to the appearance of a graph.
<-> Border	The <-> <i>Border</i> operation allows you to place or remove a border around a node.
<-> Shade	The <-> <i>Shade</i> operation allows you to change the background shade of a node. Currently, GRAPHER only supports WHITESHADE or BLACKSHADE. Thus, the operation is equivalent to inverting the node.

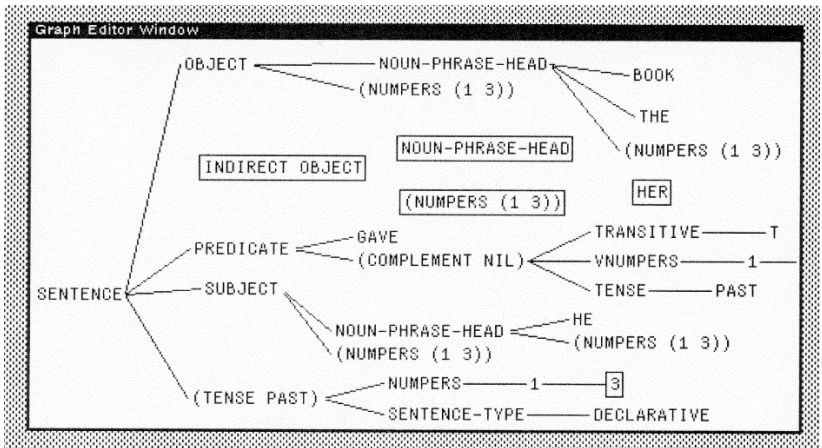


Figure 4-22. Example of Add Node

DISPLAY-ORIENTED TOOLS

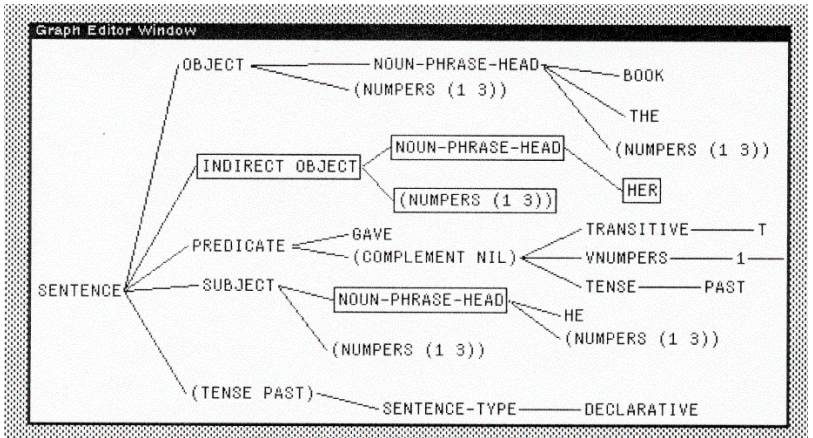


Figure 4-23. Adding a Link Example

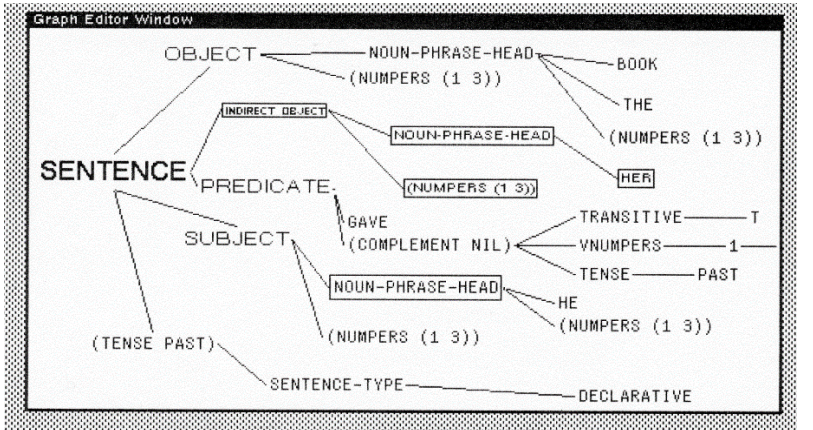


Figure 4-24. Examples of Enlarging and Reducing Labels

4.3.5.4 Handling Copy-Select Events

DISPLAY-ORIENTED TOOLS

If COPYBUTTONEVENTFN has a function as its value, this function is invoked when you attempt to copy-select an object from the Grapher window. The default function is COPYINSERT, which merely creates a Grapher object (see Section 4.3.10).

4.3.6 Laying Out a Graph

The process of creating a graph is called the "laying out" of the graph. You may layout a graph using the function LAYOUTGRAPH, which takes the form:

Function:	LAYOUTGRAPH
# Arguments:	7
Arguments:	1) NODELST, a list of nodes 2) ROOTIDS, a list of root nodes for the graph 3) FORMAT, a specifier for the geometry 4) FONT, the font descriptor 5) MOTHERD, the minimum distance between a mother and her daughters 6) PERSONALD, the minimum distance between any two nodes 7) FAMILYD, the minimum distance between two nodes of different nuclear families
Value:	A GRAPH record.

LAYOUTGRAPH "lays out" a partially specified graph by assigning positions to its graph nodes. Its result is a GRAPH record which may displayed using SHOWGRAPH. The required fields are NODELST and ROOTIDS. All other arguments are optional.

DISPLAY-ORIENTED TOOLS

NODELST is a list of partially specified graph nodes. Each node contains the node identifier, the node label, and the nodes to which links should be drawn from the current node. Consider this simple node list:

```
<-(SETQ NODELST
  (LIST
    (create GRAPHNODE
      NODELABEL <- SENTENCE
      NODEID <- 1
      TONODES <- (2 3 4))
    (create GRAPHNODE
      NODELABEL <- SUBJECT
      NODEID <- 2
      TONODES <- (5))
    (create GRAPHNODE
      NODELABEL <- PREDICATE
      NODEID <- 3
      TONODES <- (6))
    (create GRAPHNODE
      NODELABEL <- OBJECT
      NODEID <- 4
      TONODES <- (7))
    (create GRAPHNODE
      NODELABEL <- NOUN-PHRASE-HEAD
      NODEID <- 5
      TONODES <- (8))
    (create GRAPHNODE
      NODELABEL <- NOUN-PHRASE-HEAD
      NODEID <- 6
      TONODES <- (9))
    (create GRAPHNODE
      NODELABEL <- GAVE
      NODEID <- 7
      TONODES <- NIL))
```

DISPLAY-ORIENTED TOOLS

```
(create GRAPHNODE
  NODELABEL <- HE
  NODEID <- 8
  TONODES <- NIL))
(create GRAPHNODE
  NODELABEL <- (THE PARK)
  NODEID <- 9
  TONODES <- NIL))
))
((1 T NIL NIL NIL NIL NIL (2 3 4)
  NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  SENTENCE NIL)
(2 T NIL NIL NIL NIL NIL (5)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  SUBJECT NIL)
(3 T NIL NIL NIL NIL NIL (7)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  PREDICATE NIL)
(4 T NIL NIL NIL NIL NIL (6)
  (1)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  OBJECT NIL)
(5 T NIL NIL NIL NIL NIL (8)
  (2)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  NOUN-PHRASE-HEAD NIL)
(6 T NIL NIL NIL NIL NIL (9)
  (4)
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  NOUN-PHRASE-HEAD NIL)
(7 NIL NIL NIL NIL NIL NIL NIL NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
```

DISPLAY-ORIENTED TOOLS

```
GAVE NIL)
(8 NIL NIL NIL NIL NIL NIL NIL NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  HE NIL)
(9 NIL NIL NIL NIL NIL NIL NIL NIL
  (GACHA 10 (MEDIUM REGULAR REGULAR))
  (THE PARK) NIL)
)
```

All other fields normally specified in a GRAPHNODE record will be overwritten with defaults or ignored by LAYOUTGRAPH.

ROOTIDS is a list of the node identifiers of nodes that become the roots of the graph.

Now, we can layout the graph described by NODELST as follows:

```
<-(SETQ MYGRAPH (LAYOUTGRAPH NODELST '(1)))
(((1 (29 . 24)
     NIL NIL NIL 59 15 (2 3 4)
     NIL
     (GACHA 10 (MEDIUM REGULAR REGULAR))
     SENTENCE NIL)
 (2 (127 . 7)
     NIL NIL NIL 52 15 (5)
     (1)
     (GACHA 10 (MEDIUM REGULAR REGULAR))
     SUBJECT NIL)
 (3 (106 . 22)
     NIL NIL NIL 10 15 (9)
     (1)
     (GACHA 10 (MEDIUM REGULAR REGULAR))
     PREDICATE NIL)
 (4 (123 . 41)
```

DISPLAY-ORIENTED TOOLS

NIL NIL NIL 45 15 (6)
(1)
(GACHA 10 (MEDIUM REGULAR REGULAR))
OBJECT NIL)
((5 (252 . 7)
NIL NIL NIL 115 15 (7)
(2)
(GACHA 10 (MEDIUM REGULAR REGULAR))
NOUN-PHRASE-HEAD NIL)
((6 (245 . 41)
NIL NIL NIL 115 15 (8)
(4)
(GACHA 10 (MEDIUM REGULAR REGULAR))
NOUN-PHRASE-HEAD NIL)
((7 (360 . 7)
NIL NIL NIL 17 15 NIL
(5)
(GACHA 10 (MEDIUM REGULAR REGULAR))
GAVE NIL)
((8 (381 . 41)
NIL NIL NIL 73 15 NIL
(6)
(GACHA 10 (MEDIUM REGULAR REGULAR))
HE NIL)
((9 (168 . 22)
NIL NIL NIL 31 15 NIL
(3)
(GACHA 10 (MEDIUM REGULAR REGULAR))
(THE PARK) NIL))
T NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)

NODEFONT is the font specification for printing the node labels in the graph.

4.3.6.1 Controlling Formatting

The FORMAT argument controls the geometry of the graph, e.g., how the graph will be oriented with respect to the window it is displayed in. It is a list of atoms. There are four basic formats as presented in Table 4-11.

Table 4-11. Basic Formats

Format	Description
COMPACT	<p>This is the default format. The graph will be laid out, if possible, as a forest (e.g., a set of trees) such that the amount of screen space utilized is minimized. Figure 4-25 depicts a compact graph generated by the following expression:</p> <pre><-(SHOWGRAPH (LAYOUTGRAPH NODELST ROOTIDS '(COMPACT))) {WINDOW}#74,25000</pre>
FAST	<p>The graph is laid out as a forest, but screen space is sacrificed for speed of display. For large, complex graphs this is recommended due to the computations necessary to generate the graph. Figure 4-26 depicts a graph laid out in FAST format as generated by the following expression:</p> <pre><-(SHOWGRAPH (LAYOUTGRAPH NODELST ROOTIDS '(FAST))) {WINDOW}#74,25000</pre>
LATTICE	<p>The graph is laid out as an acyclic directed graph. Figure 4-27 depicts a graph laid out as a lattice as generated by the following expression:</p> <pre><-(SHOWGRAPH (LAYOUTGRAPH NODELST ROOTIDS '(LATTICE))) {WINDOW}#74,25000</pre>
REVERSE	<p>The graph is laid out with the daughters (e.g., leaves) laid out at the top of the graph. Figure 4-28 depicts a</p>

DISPLAY-ORIENTED TOOLS

	<p>graph laid out in REVERSE format as generated by the following expression:</p> <pre style="margin: 0;"><-(SHOWGRAPH (LAYOUTGRAPH NODELST ROOTIDS '(REVERSE))) {WINDOW}#74,25000</pre>
--	---

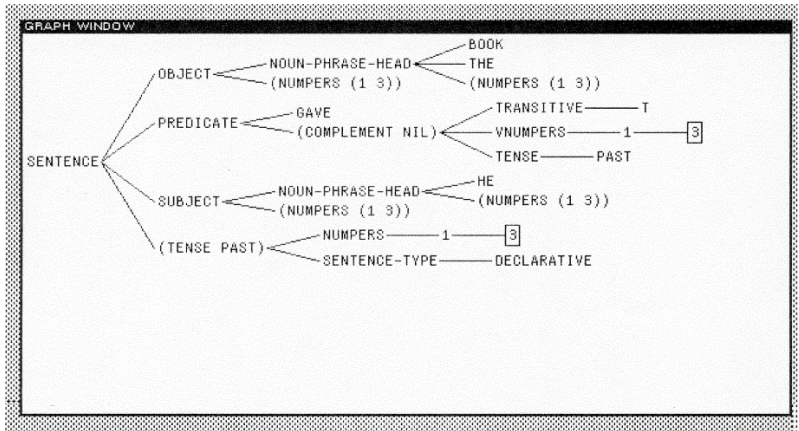


Figure 4-25 A Graph laid out in COMPACT Format

DISPLAY-ORIENTED TOOLS

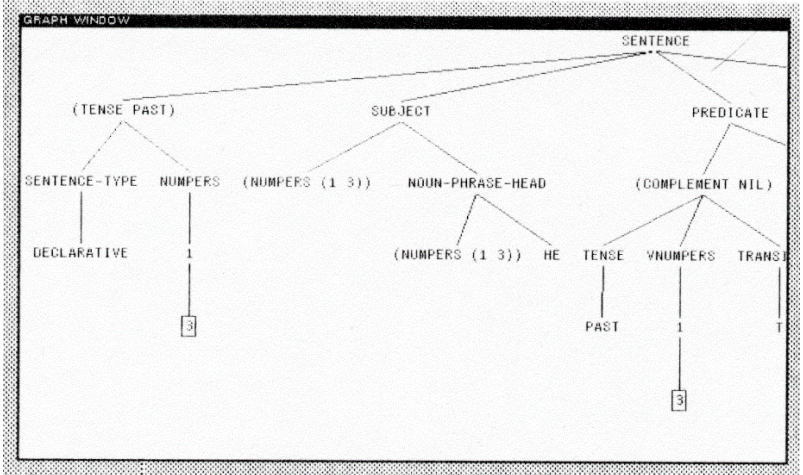


Figure 4-26. A graph laid out in FAST format

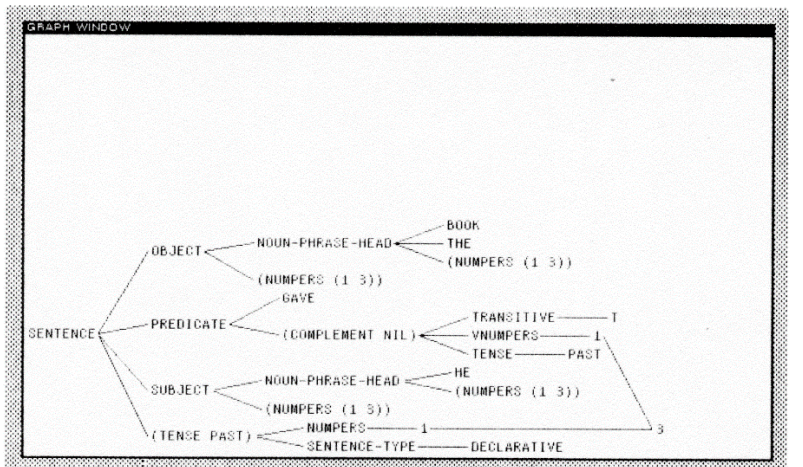


Figure 4-27. A graph laid out in LATTICE format

DISPLAY-ORIENTED TOOLS

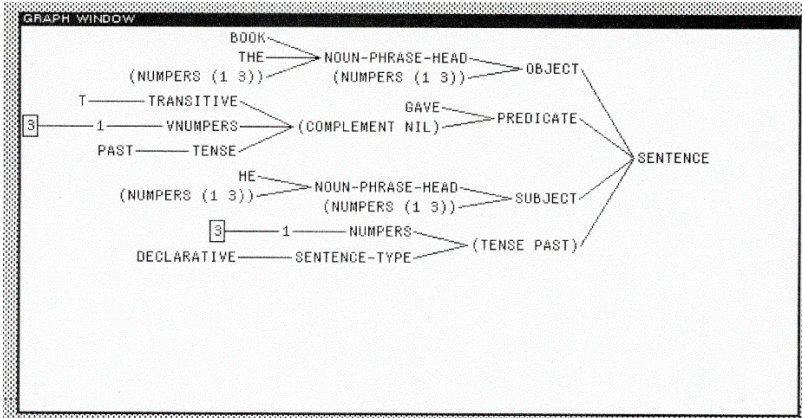


Figure 4-28. A Graph laid out in REVERSE format

4.3.6.2 Controlling Graph Orientation

The orientation of the graph with respect to the window may be specified by the following atoms in Table 4-12 included after the formatting specification.

Table 4-12. Graph Orientation Values

Orientation	Description
HORIZONTAL	<p>The roots of the graph are depicted at the left edge of the window and the links run to the right. Figure 4-29 depicts a horizontal compact graph generated by the following expression:</p> <pre>(SHOWGRAPH (LAYOUTGRAPH NODELST ROOTIDS '(COMPACT HORIZONTAL))) {WINDOW}#74,25000</pre>
VERTICAL	<p>The roots of the graph appear at the top of the window and the links run from the top to the bottom of the window. Figure 4-30 depicts a</p>

DISPLAY-ORIENTED TOOLS

	<p>vertical compact graph generated by the following expression:</p> <p style="text-align: center;">(SHOWGRAPH (LAYOUTGRAPH NODELST ROOTIDS '(COMPACT VERTICAL))) {WINDOW}#74,25000</p>
--	---

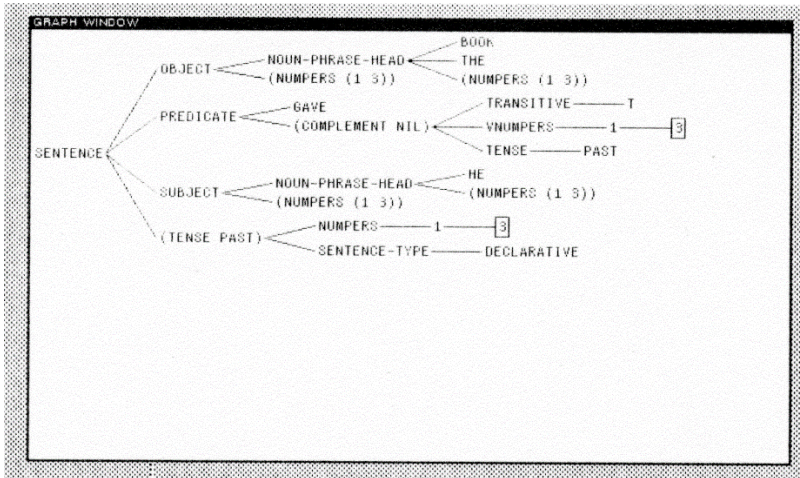


Figure 4-29 Example of a COMPACT HORIZONTAL graph

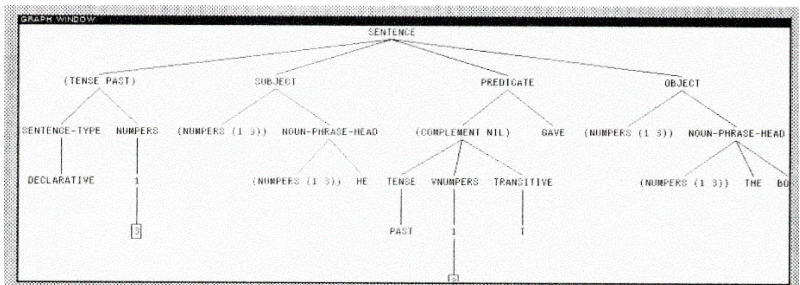


Figure 4-30. Example of a COMPACT VERTICAL graph

The directions of the links may be reversed by including the atom REVERSE in the format specification.

4.3.6.3 Virtual Nodes

In complex graphs with many links from one node to other nodes, the graph may become rather "messy". To avoid this situation and make the graph clearer, LAYOUTGRAPH will create *virtual nodes* in order to consolidate many links going to a single node. When it does so, NODELST is modified to include the virtual nodes and the TONODES fields are modified appropriately.

Whether or not LAYOUTGRAPH creates virtual nodes depends upon the inclusion of LATTICE in the format specification. The algorithms for the two cases are sketched out in Section 4.3.6.4 and 4.3.6.5, respectively.

4.3.6.4 Formatting Forests

When Grapher formats a forest, it first lays out the nodes by traversing the forest top-down using a depth-first traversal algorithm. When it finds a node which has already been assigned a node position, it does not draw a link that might cross other nodes or links. Rather, it creates a copy of the node, which has no TONODE elements, and inserts it in the graph. It marks the original node and the virtual node by setting their NODEBORDER and NODELABELSHADE fields. Thus, a *marked* node occurs at least twice in a forest.

You may control the appearance of marked nodes by adding the atom MARK to the FORMAT argument. Its tail is a property list which is interpreted according to Table 4-13.

Table 4-13. MARK Format Values

Format Value	Description
NIL	No special marking is performed.
BORDER	The corresponding value is used in the NODEBORDER field of the marked nodes.
LABELSHADE	The corresponding value is used in the NODELABELSHADE field of the marked nodes.
COPIES/ONLY	Only new virtual nodes are marked; the original nodes are left unmarked.
NOT/LEAVES	Nodes which have no daughters will not be specially marked.

4.3.6.5 Formatting Lattices

If the format specification includes the atom LATTICE, then virtual nodes which are the daughter of more than one node are not marked. Instead, links from all of its parents are drawn to it. No attempt is made to avoid drawing lines through nodes or to minimize line crossings.

In HORIZONTAL format, nodes will be positioned in the graph so that From nodes always appear to the left of To nodes.

In VERTICAL format, the TONODES of a node are always positioned beneath it and the FROMNODES are positioned above it.

Note, however, that Grapher cannot draw cyclic graphs using this convention. Thus, whenever Grapher detects a node that points to itself, it creates a virtual node and marks both the original and the

copy (subject to the format specification as described above).

4.3.6.6 Distance Specifications

Grapher allows you to specify the distance between pairs of nodes according to the three criteria presented in Table 4-14.

Table 4-14. Node Distance Criteria

Criterion	Description
MOTHERD	Specifies the minimum distance between a mother and her daughter nodes.
PERSONALD	Specifies the minimum distance between any two nodes.
FAMILYD	Specifies the minimum distance between two nodes different parents, but common grandparents.

The closest that two nodes will appear in a graph if they are not sisters is (PLUS PERSONALD FAMILYD).

4.3.7 Editing a Graph

You may edit a graph using the function **EDITGRAPH**, which takes the form:

Function:	EDITGRAPH
# Arguments:	2
Arguments:	1) GRAPH, a list representing the graph 2) WINDOW, a window handle
Value:	The list representing the graph, as edited.

DISPLAY-ORIENTED TOOLS

EDITGRAPH displays the nodes of the graph in the window. The conditions concerning graphs and windows operate as described in SHOWGRAPH.

EDITGRAPH assigns specific functions to the left and middle buttons of the mouse which are described in Section 4.3.5.3. Figure 4-31 depicts the EDITGRAPH menu.

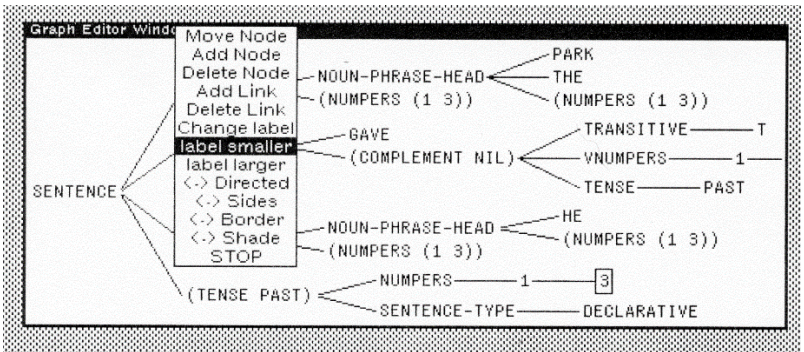


Figure 4-31. The EDITGRAPH Menu

4.3.7.1 EDITGRAPH1

EDITGRAPH1 is the workhorse function that is used by EDITGRAPH. It takes the form:

Function:	EDITGRAPH1
# Arguments:	1
Arguments:	1) WINDOW, a window handle
Value:	NIL

EDITGRAPH1 assumes that WINDOW already has a graph displayed in it. It accesses the list representing the graph from the GRAPH property of the window handle. It allows you to edit a graph displayed in a window.

4.3.8 Laying Out a Forest

A *forest* is a graph which is represented as a set of independent trees. To layout a forest in a window, you may use the function **LAYOUTFOREST**, which takes the form:

Function:	LAYOUTFOREST
# Arguments:	8
Arguments:	<ol style="list-style-type: none"> 1) NODELST, a list of nodes 2) ROOTIDS, a list of nodes which are the roots of the trees 3) FORMAT, a format indicator 4) BOXING, a flag 5) FONT, a font descriptor 6) MOTHERD, minimum distance to mother node 7) PERSONALD, minimum distance between any two nodes 8) FAMILYD, the minimum distance between two nodes of different nuclear families
Value:	A graph record.

LAYOUTFOREST lays out a graph as a set of trees (e.g., a forest). It uses boxing to avoid drawing a set of lines which collide when the set of trees is not actually a forest. It produces a graph record which can be displayed with SHOWGRAPH.

LAYOUTFOREST requires, at a minimum, a node list - NODELIST, which is a list of GRAPHNODES - and a list of roots - ROOTIDS, which are the names of nodes which are the roots of the individual trees. The other arguments are similar to those described for LAYOUTGRAPH.

DISPLAY-ORIENTED TOOLS

Nodes are laid out by traversing the forest top-down, depth-first. Whenever a node that has already been laid out is encountered, **LAYOUTFOREST** creates a copy of the node rather than drawing a link to it that might cause it to cross intersecting lines. Hence, a box around a node indicates that it has occurred at least twice in the forest.

BOXING allows you to modify this basic strategy somewhat. It takes one of the values in Table 4-15.

Table 4-15. Boxing Values

Boxing Value	Description
COPIES/ONLY	Only copies nodes are boxed; the original nodes are left unmarked.
NOT/LEAVES	Nodes which have no daughters will not be specially boxed.

Thus, the choice **BOXING = (COPIES/ONLY NOT/LEAVES)** boxes nodes that are copies of nodes that have daughters (i.e., if you see a box around a node, the node has daughters that aren't drawn).

4.3.9 Laying Out an S-Expression

LAYOUTGRAPH takes a list of graph nodes as its argument. It is time-consuming and tedious to construct these graph nodes. Grapher provides an alternative function, **LAYOUTSEXPR**, which takes an S-expression as its argument and creates a graph record. It takes the form:

Function:	LAYOUTSEXPR
# Arguments:	7
Arguments:	1) SEXPR, an S-expression 2) FORMAT, a format specification for the geometry

DISPLAY-ORIENTED TOOLS

- 3) BOXING, a flag indicating node boxing
 - 4) FONT, the font descriptor
 - 5, MOTHERD, the minimum distance between a mother and her daughters
 - 6) PERSONALD, the minimum distance between any two nodes
 - 7) FAMILYD, the minimum distance between two nodes of different nuclear families
- Value: A GRAPH record.

LAYOUSEXPR interprets its arguments in a manner similar to LAYOUTGRAPH.

SEXPR is recursively interpreted as follows. If SEXPR is not a list, its NODELABEL is itself and it has no TONODES (e.g., it is a single node graph). Otherwise, the CAR of SEXPR is assumed to be the NODELABEL and its CDR is taken as a list of TONODES. Thus, the CDR must be a list of S-expressions. Circular S-expressions are allowed.

Consider the following example in which a partial graph record has been printed out.

```
<-(SETQ graph (LAYOUTSEXPR PARSE))
(((THE)
 (1099 . 122)
  NIL NIL NIL 26 17 NIL
  ((NOUN-PHRASE-HEAD ((NUMPERS (1 3))) THE PARK))
   {FONTCLASS}#70,172764 THE T)
 (3)
 (496 . 8)
  NIL NIL NIL 12 17 NIL
```

```

((1 3))
 {FONTCLASS}#70,172764 3 T)
(PARK
 (1141 . 123)
 NIL NIL NIL 31 15 NIL
 ((NOUN-PHRASE-HEAD ((NUMPERS (1 3))) THE PARK))
 {FONTCLASS}#70,172764 PARK NIL)
(((NUMPERS (1 3)))
 (1018 . 123)
 NIL NIL NIL 108 15 NIL
 ((NOUN-PHRASE-HEAD ((NUMPERS (1 3)) THE PARK))
 {FONTCLASS}#70,172764 (NUMPERS (1 3)) NIL)
 ...

```

4.3.10 Creating an Image Object for a Graph

You may create an image object that encapsulates a graph using the function **GRAPHEROBJ**, which takes the form:

Function:	GRAPHEROBJ
# Arguments:	3
Arguments:	1) GRAPH, a graph record 2) HALIGN, an horizontal alignment 3) VALIGN, a vertical alignment
Value:	An image object handle.

GRAPHEROBJ creates an image object that can display GRAPH. The arguments HALIGN and VALIGN specify how the graph is to be aligned with respect to some reference point in the host stream in which it is displayed. Consider the following example:

```

<-(GRAPHEROBJ vertgraph)
 {IMAGEOBJ}#74,36306

```

The OBJECTDATUM field of the image object contains the graph structure.

HALIGN and VALIGN take values between 0 and 1. These values specify the proportion of the width and height of the graph will overlay its reference point. A value of 0 means the graph is displayed to the left and above of the reference point, while a value of 1 means that it will be displayed to the right and below. They can also take a pair of values of the form

(<node-specification> <position>)

where <node-specification> is the node identifier of the graph that it is to be aligned by and <position> is a location in the node where the alignment point is.

4.3.10.1 Specifying a Frontier Node

The <node-specification> may also be one of the atoms: *TOP*, *BOTTOM*, *LEFT*, *RIGHT*, which indicates the node of the graph to use for alignment.

4.3.10.2 Specifying the Position

The <position> may be a floating-point number which specifies a proportional distance from the lower left corner of the node. Or, it may be the atom BASELINE which indicates the character nearest the baseline.

4.3.11 Determining the Minimal Graph Region

You may use **GRAPHREGION** to determine the smallest region which can contain all nodes of a graph when they are laid out according to the specifications given LAYOUTGRAPH. It takes the

form:

Function:	GRAPHREGION
# Arguments:	1
Arguments:	1) GRAPH, a graph structure
Value:	A region specification.

GRAPHREGION computes the smallest region that will contain all of the nodes of the graph when fully displayed. This function is useful when you are deciding where and how to display a graph. You may create a graph using LAYOUTGRAPH and then determine the region necessary to display. Using this value, you can determine the size of the window that you want to open to display the graph. Consider the example:

```
<-(GRAPHREGION vertgraph)
(-7 0 908 302)
```

```
<-(GRAPHREGION horizgraph)
(0 0 531 183)
```

4.3.12 Inverting a Graph Node

When you select a graph node, Grapher inverts the node and a local region to indicate that it has been selected. You may use **FLIPNODE** to invert graph nodes that are selected inside user-provided editing functions. It takes the form:

Function:	FLIPNODE
# Arguments:	2
Arguments:	1) NODE, a graph node 2) WINDOW/STREAM, a window or display stream handle
Value:	T.

DISPLAY-ORIENTED TOOLS

FLIPNODE inverts a region around the specified node that is one pixel larger than the image of the node itself. Consider the following example:

```
<-(SETQ mynode (CADAR mygraph))
(BOOK (360 . 175)
      NIL
      NIL
      NIL
      31 15
      NIL
      ((NOUN-PHRASE-HEAD
        ((NUMPERS (1 3)))
        THE BOOK))
      {FONTCLASS}#70,172764 BOOK NIL)
```

```
<-(FLIPNODE mynode iw)
T
```

4.3.13 Resetting the Node Border

You may reset the border (e.g., size) of a node using the function **RESET/NODE/BORDER**, which takes the form:

Function:	RESET/NODE/BORDER
# Arguments:	4
Arguments:	1) NODE, a graph node 2) BORDER, an integer or 'INVERT 3) WINDOW/STREAM, a window or display stream handle 4) GRAPH, a graph structure
Value:	The graphnode record.

This function changes the border of the node of the graph which is displayed in the window associated with WINDOW/STREAM.

DISPLAY-ORIENTED TOOLS

Changing the node border may change the size of the node. If it does, the lines leading to and from the node will be redrawn. This may also change the position of the node in the graph. Because the node position and line positions are computed as a function of the entire graph, the graph structure must be made available to this function. Consider the following examples:

```
<-(RESET/NODE/BORDER  mynode
                        'INVERT
                        IW
                        mygraph)
(BOOK (360 . 175)
      NIL
      NIL
      NIL
      31 15
      NIL
      ((NOUN-PHRASE-HEAD
        ((NUMPERS (1 3)))
        THE BOOK))
      FONTCLASS}#70,172764
      BOOK
      T
)
```

The value of BORDER may be the atom INVERT which causes the border to be displayed in an inverted form.

4.3.14 Resetting the Node's Label Shade

You may change the shade with which the label of a node is displayed using the function RESET/NODE/LABELSHADE, which takes the form:

Function: RESET/NODE/LABELSHADE

DISPLAY-ORIENTED TOOLS

Arguments: 3
Arguments: 1) NODE, a graph node
2) SHADE, a shade specification
3) WINDOW/STREAM, a window or display stream handle
Value: The graphnode record.

Often, you will want to change the node's label shade to indicate some special condition about the node, such as an active node in a search or traversal of the graph. Consider the following example:

```
<-(RESET/NODE/SHADE mynode GRAYSHADE IW)
(BOOK (360 . 175)
  NIL
  NIL
  43605
  31 15
  NIL
  ((NOUN-PHRASE-HEAD
  ((NUMPERS (1 3))
  THE BOOK))
  {FONTCLASS}#70,172764
  BOOK
  T
)
```

SHADE may take the atom INVERT which causes the node's label shade to be inverted.

4.3.15 Dumping a Graph to a File

In order to save a graph on a file, you should use the function **DUMPGRAPH**, which takes the form:

Function: DUMPGRAPH
Arguments: 2

DISPLAY-ORIENTED TOOLS

Arguments: 1) GRAPH, a graph specification
 2) STREAM, a display stream handle
Value: A compact form of the graph.

When graphs become complex, the graph specification can grow exponentially because of the number of linkages that must be represented among the nodes. DUMPGRAPH prints the graph on STREAM in a compact form which can be read by READGRAPH (see below).

The IRM notes that a graph cannot be saved on a file using the standard print functions because the Grapher uses FASSOC to fetch a graph node using its identifier. You could use HPRINT, but it dumps a total description of the node each time it is mentioned. DUMPGRAPH rectifies this problem by compacting the specification in a form that is easily understood by READGRAPH. Consider the following example:

```
<-(OPENFILE 'SHKGRAPH.TXT 'OUTPUT 'NEW)
{DSK}<LISPFILES>SHKGRAPH.TXT;1
```

```
<-(DUMPGRAPH vertgraph 'SHKGRAPH.TXT)
NIL
```

```
<-(CLOSEF 'SHKGRAPH.TXT)
{DSK}<LISPFILES>SHKGRAPH.TXT;1
```

Now, we can inspect the contents of the file SHKGRAPH.TXT by sending it to the printer. A portion of the contents appears as:

```
(  FIELDS ()
   IDS 28 (3)
   PARK THE ((NUMPERS (1 3)))
             (NOUN-PHRASE_HEAD ((NUMPERS (1 3))))
   THE PARK)
```

DISPLAY-ORIENTED TOOLS

```
((NUMPERS (1 3)))
  (OBJECT ((NUMPERS (1 3)))
    (NOUN-PHRASE-HEAD
      ((NUMPERS (1 3)))
      THE PARK))
GAVE T (TRANSITIVE T) (1 3) (VNUMPERS (1 3)) PAST
  (TENSE PAST) ((COMPLEMENT NIL) (TENSE PAST) ... )
  FONTS 1 (CLASS DEFAULTFONT 1 (GACHA 10)
    (GACHA 8) (TERMINAL 8) (4045XLP (TITAN 10)))
  NODES (
    (1 3
      (496 . 8) 1 T NIL NIL (11))
    (2 PARK
      (885 . 123) 1 NIL NIL NIL (5))
    (3 THE
      (844 . 123) 1 NIL NIL NIL (5))
    (4 (NUMPERS (1 3))
      (764 . 123) 1 NIL NIL NIL (5))
    (5 NOUN-PHRASE-HEAD
      (824 . 180) 1 NIL NIL (4 3 2) (7))
    (6 (NUMPERS (1 3))
      (699 . 180) 1 NIL NIL NIL (7))
    (7 OBJECT
      (761 . 237) 1 NIL NIL (6 5) (28))
    ...
```

Note that there four major descriptors in the file: **FIELDS**, **IDS**, **FONTS**, and **NODES**. This compact form serves to describe the entire graph so that **READGRAPH** can recreate the graph structure (see below).

4.3.16 Reading a Graph from a File

You may read a description of a graph from a file or stream using the function **READGRAPH**, which takes the form:

DISPLAY-ORIENTED TOOLS

Function:	READGRAPH
# Arguments:	1
Arguments:	1) STREAM, a display stream handle
Value:	A graph specification.

READGRAPH reads the information necessary to reconstruct a graph specification from STREAM. It starts at the current file pointer. It returns a graph structure equivalent to the one compacted by DUMPGRAPH. Consider the following example:

```
<-(OPENFILE 'SHKGRAPH.TXT 'INPUT 'OLD)
{DSK}<LISPFILES>SHKGRAPH.TXT;1

<-(SETQ MYGRAPH (READGRAPH 'SHKGRAPH.TXT))
(((3)
 (496 . 8)
  NIL NIL NIL NIL NIL NIL
  ((1 3))
   {FONTCLASS}#70,172,740 3 T)
 (PARK
  (885 . 123)
  NIL NIL NIL NIL NIL NIL
  ((NOUN-PHRASE-HEAD ((NUMPERS (1 3))) THE PARK))
   {FONTCLASS}#70,172740 PARK NIL)
 (THE
  (844 . 123)
  NIL NIL NIL NIL NIL NIL
  ((NOUN-PHRASE-HEAD ((NUMPERS (1 3))) THE PARK))
   {FONTCLASS}#70,172740 THE NIL)
 (((NUMPERS (1 3)))
  (764 . 123)
  NIL NIL NIL NIL NIL NIL
  ((NOUN-PHRASE-HEAD ((NUMPERS (1 3))) THE PARK))
   {FONTCLASS}#70,172740 (NUMPERS (1 3)) NIL)
```

DISPLAY-ORIENTED TOOLS

...

5. Graphics

Because Interlisp operates in a bitmapped display environment, it can provide a powerful graphics facility for drawing pictures, charts, and other diagrams. Interlisp also provides a few graphics routines. Using these routines, you can build a powerful graphics programming environment.

5.1 Basic Concepts

I have created the following image stream to be used in the various examples:

```
<-(SETQ IMS  
  (OPENIMAGESTREAM "Sample Image Stream"  
    '(500 10 500 250))  
{STREAM}#65,16320
```

Thus, I generally will not show the creation of an image stream in the following examples, but merely refer to IMS.

Image streams were discussed in Chapter 7. An image stream is the basic destination for all graphics operations.

5.1.1 Brushes

For certain drawing functions, Interlisp provides different types of brushes for *painting* the pattern on the display stream. The types of brushes are:

1. ROUND
2. SQUARE
3. HORIZONTAL

Graphics

4. VERTICAL
5. DIAGONAL

Figure 5-1 depicts an example of a round brush.

```
<-(SET.DSP.POSITION 100 100 awindow)
(100 . 100)
```

Draw a curve using a round brush with dashing:

```
<-(DRAWCURVE '((100 . 100)
               (100 . 150)
               (50 . 275)
               (250 . 345))
              T
              '(ROUND 3)
              '(5 5)
              awindow)
```

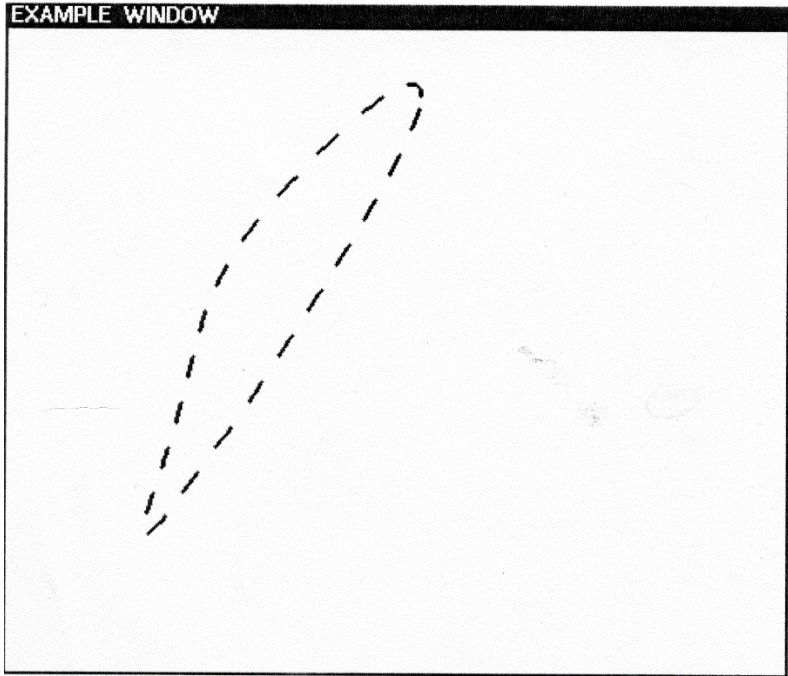


Figure 5-1. An Example of a ROUND Brush with dashing

A brush is specified as a three element list whose form is:

(<shape> <width> <color>)

where: <shape> is one of the brush types
<width> is the number of pixels
comprising the brush
<color> is the color of the brush

The default brush type is (ROUND 1). If BRUSH is specified as NIL in any of the drawing functions, the default brush type will be

Graphics

used. A brush type may also be specified as an integer (such as "3"), which is interpreted as a round brush of width 3 pixels.

5.1.1.1 Installing a New Brush

You may define new brush shapes using the function **INSTALLBRUSH**, which takes the form:

Function:	INSTALLBRUSH
# Arguments:	3
Arguments:	1) BRUSHNAME, the name of the brush 2) BRUSHFN, a brush function defining its bitmap 3) BRUSHARRAY, a hand-crafted brush description (optional)
Value:	NIL.

The brush name us is added to the list of brushes known to the system. BRUSHFN specifies the bitmap which is used when the brush is used.

5.1.1.2 Drawing a Brush

You can draw an image of a brush at a point using the function **DRAWPOINT**, which takes the form:

Function:	DRAWPOINT
# Arguments:	5
Arguments:	1) X, an X coordinate 2) Y, a Y coordinate 3) BRUSH, a brush specification 4) STREAM, a display stream handle 5) OPERATION, a bitmap operation

Graphics

Value: T

DRAWPOINT draws an image of the brush at the point (X,Y) in STREAM using the specified operation. BRUSH may be a bitmap or a brush. The default operation is PAINT. Consider the following example:

```
<-(DRAWPOINT 100 100 '(ROUND 50) IMS 'PAINT)  
T
```

I made the brush 50 pixels wide so that it would be easy to see in the figure.

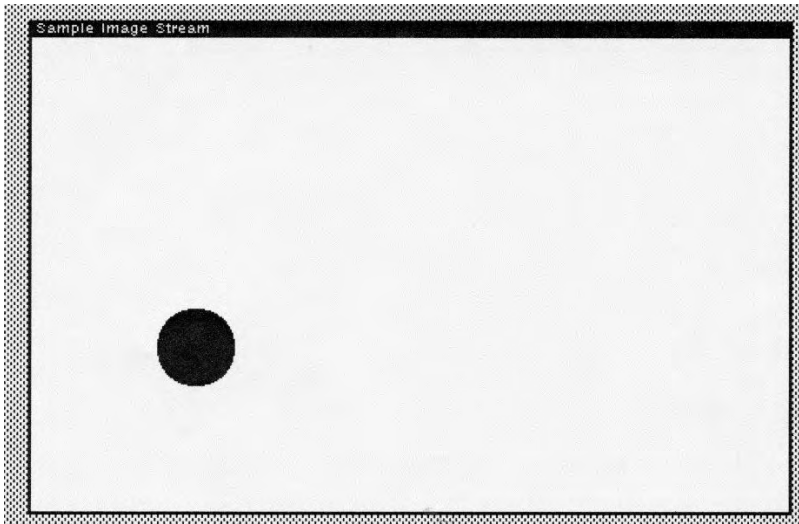


Figure 5-2. Example of a ROUND Brush

The center of the brush is located at the point (X,Y).

Now, here is an example of a square brush.

Graphics

```
<-(DRAWPOINT 100 100 '(SQUARE 50) IMS 'PAINT)  
T
```

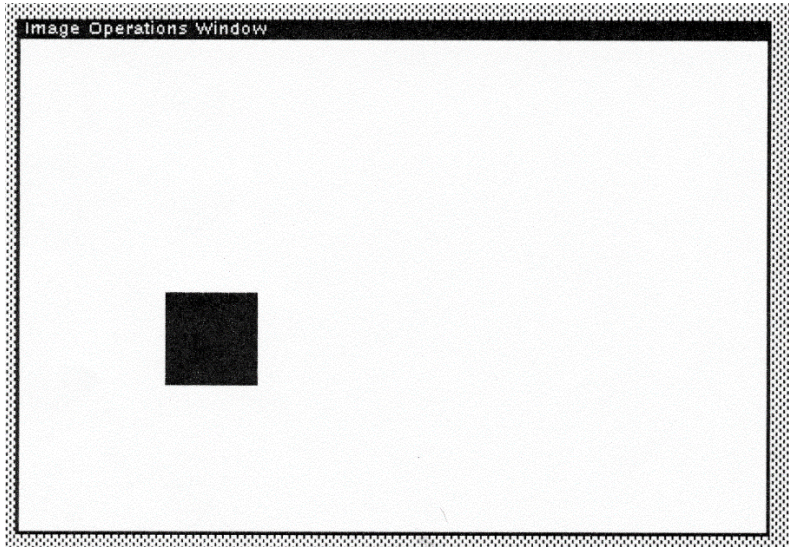


Figure 5-3. Example of a SQUARE Brush

5.1.1.3 Reading the Brush Shape

You can prompt a user to select a brush shape using the function **PAINW.READBRUSHSHAPE**, which takes the form:

Function:	PAINW.READBRUSHSHAPE
# Arguments:	0
Arguments:	N/A
Value:	A brush shape.

PAINW.READBRUSHSHAPE displays a menu of the currently installed brush shapes at the current location of the cursor. You may select one of the brush shapes by pointing to it and clicking the left

Graphics

mouse button. Alternatively, NIL will be returned by clicking any mouse button outside of the menu. The initial form of the menu is:

```
DIAGONAL
VERTICAL
HORIZONTAL
SQUARE
ROUND
```

Consider the following example:

```
<-(PAINTW.READBRUSHSHAPE)
VERTICAL
```

where I pointed at the "vertical" item in the menu and clicked the left mouse button.

5.1.1.4 Reading the Brush Size

You can prompt the user to select a brush size using the function **PAINTW.READBRUSHSIZE**, which takes the form:

Function:	PAINTW.READBRUSHSIZE
# Arguments:	0
Arguments:	N/A
Value:	A brush size.

PAINTW.READBRUSHSIZE displays the menu show below and waits for you to select one of the entries. Moving the mouse off the menu and clicking the left mouse button will return the value NIL. The valid brush sizes are 1,2,4,8, and 16. Consider the following example:

```
<-(PAINTW.READBRUSHSIZE)
16
```

Graphics

where I pointed at the "16" item in the menu and clicked the left mouse button.

5.1.1.5 Reading a Brush Shade

You can prompt the user to select a brush shade using the function **PAINW.READBRUSHSHADE**, which takes the form:

Function:	PAINW.READBRUSHSHADE
# Arguments:	0
Arguments:	N/A
Value:	A brush shade.

PAINW.READBRUSHSHADE displays a menu of the standard brush shades at the current location of the cursor. You may select one of the brush shades by pointing to it and clicking the left mouse button. Alternatively, NIL will be returned by clicking any mouse button outside of the menu. The initial form of the menu is depicted in Figure 5-4.

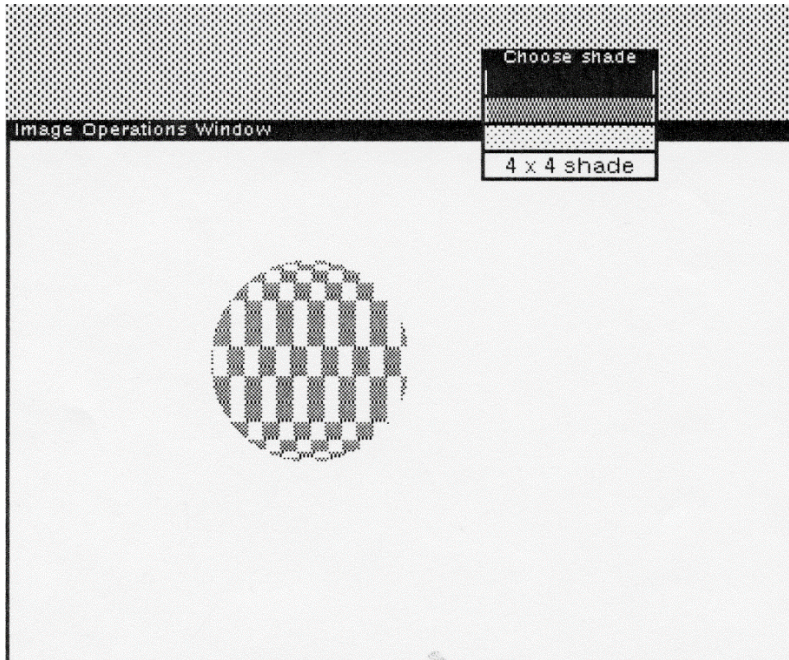


Figure 5-4. Menu of Brush Shades

Consider the following example:

```
<-(PAINTW.READBRUSHSHADE)  
43605
```

which corresponds to the numerical value of GRAYSHADE which I selected with the mouse.

Choosing the 4x4 shade opens a bitmap window as depicted in Figure 5-5. You may set pixels of the 4x4 region. When you have tailored your shade, select QUIT to exit. Figure 5-6 depicts a tailored shade whose value is 17417.

Graphics

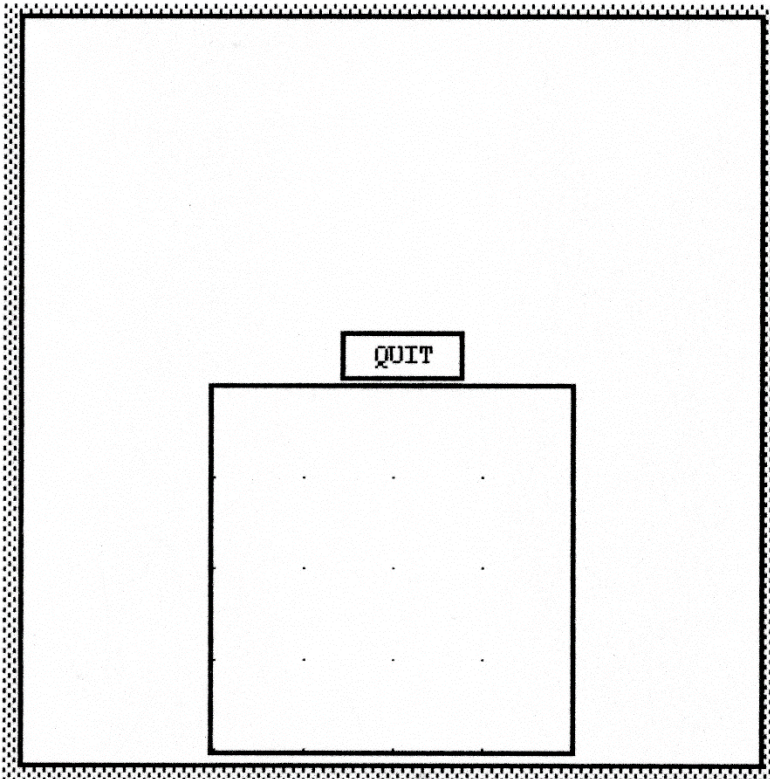


Figure 5-5. Tailoring a 4x4 Shade

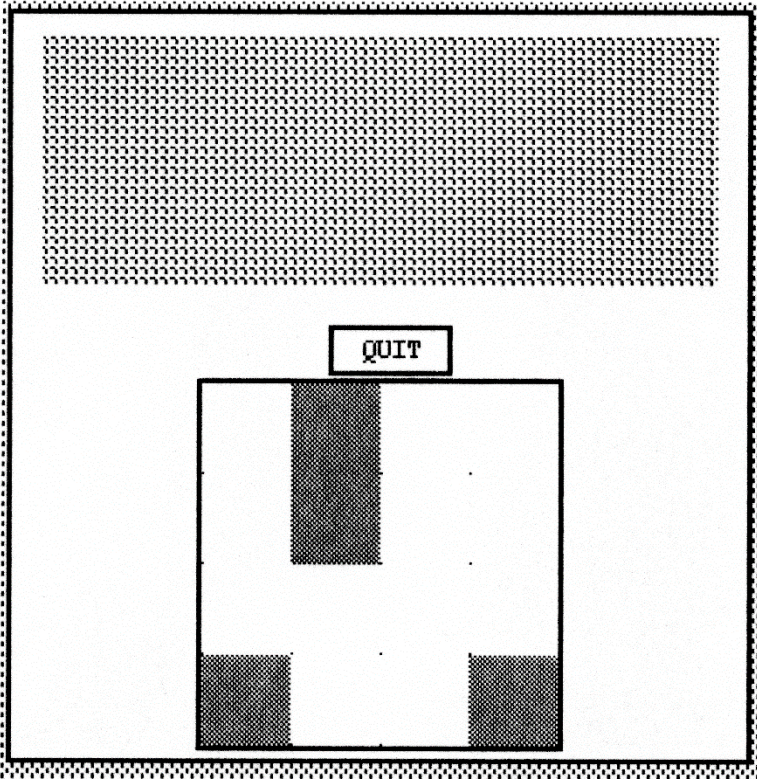


Figure 5-6. The Tailored Shade

5.1.1.6 Getting a Brush Bitmap

You can create a bitmap with an image of the appropriate brush in it using the function `\BRUSHBITMAP`, which takes the form:

Function:	<code>\BRUSHBITMAP</code>
# Arguments:	2
Arguments:	1) BRUSHSHAPE, the shape of the brush

Graphics

2) BRUSHWIDTH, the width of the brush

Value: A bit map handle.

This function creates a bit map in which is centered a brush image of the appropriate type and size. The bit map is sized to the width of the brush. You may then edit the bit map to create new brushes. Consider the following example which I have edited to create a checkered brush:

```
<-(BRUSHBITMAP 'ROUND 20)  
{BITMAP}#61,166360
```

The checkered brush is depicted in Figure 5-7.

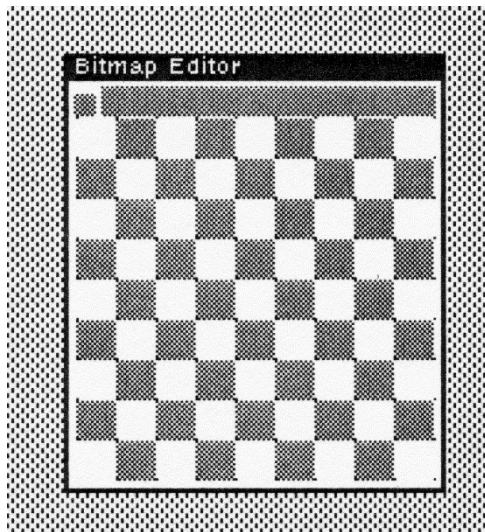


Figure 5-7. A Checkered Brush

Graphics

5.1.2 Operations

A limited number of operations are supported by the drawing functions:

1. Only the PAINT operation is supported by most types of image streams.
2. Most display streams will accept the INVERT operation for curve drawing when the brush is (ROUND 1).
3. When the brush size is larger than 1, most curve drawing operations use the ERASE operation rather than the INVERT operation.
4. For curve drawing operations on display streams, the REPLACE operation is treated the same as the PAINT operation.

5.1.3 Dashing

The DASHING argument in the following functions is a list of positive integers that determines the dashing characteristics of the line being drawn. A *dashing specification* is composed of pairs of integers of the form <dash-size> <gap-size> where:

<dash-size>	Indicates the number of pixels the brush will paint on the display stream
<gap-size>	Indicates the number of pixels the brush will not paint on the display stream

These pairs are repeated to indicate variability in the types of dashed lines that you may wish to draw.

The dashing sequence is repeated from the beginning when the list is exhausted until the drawing function is completed. If DASHING is NIL, the line to be drawn is not dashed.

Graphics

5.2 Lines and Curves

The primitive graphics functions provided by Interlisp allow you to draw lines and curves. The target of each function is a display stream or a bitmap.

5.2.1 Drawing a Line

Interlisp provides four functions for drawing a straight line between two points. They differ in how the starting point for the line is specified.

5.2.1.1 From the Current Position

DRAWTO draws a line from the current position specified of the destination bitmap to the given position. It takes the form:

Function:	DRAWTO
# Arguments:	7
Arguments:	1) X, an X-axis coordinate 2) Y, a Y-axis coordinate 3) WIDTH, the width of the line 4) OPERATION, a bitblt operation 5) DISPLAYSTREAM, a display stream or bitmap 6) COLOR, the color of the line 7) DASHING, a dashing specification
Value:	The old Y-axis coordinate.

DRAWTO draws a line from the current position of the destination bitmap to the point (X,Y). The current position of **DISPLAYSTREAM** is set to (X,Y). Consider the following example, which demonstrates several calls to **DRAWTO**:

Graphics

```
<-(SETQ XW (CREATEW))  
{WINDOW}#56,43000
```

Now, set the starting point for the drawing operations:

```
<-(SET.DSP.POSITION 100 100 XW)  
(100 . 100)
```

Draw a straight line from (100 . 100) to (200 . 200) using the default brush:

```
<-(DRAWTO 200 200 'PAINT (WINDOWPROP XW 'DSP))  
100
```

And, draw the following lines to complete a triangle, which is depicted in Figure 5-8:

```
<-(DRAWTO 100 300 'PAINT (WINDOWPROP XW 'DSP))  
200  
<-(DRAWTO 100 100 'PAINT (WINDOWPROP XW 'DSP))  
300
```

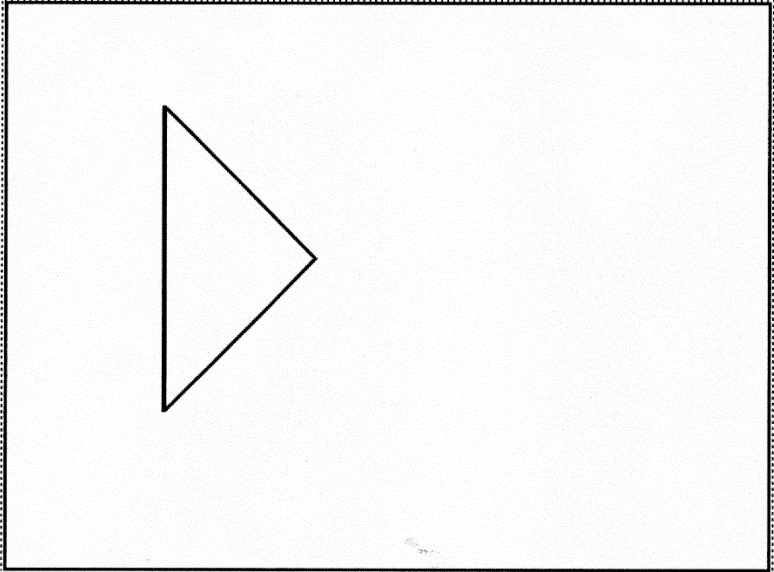


Figure 5-8. A DRAWTO Example

If the destination bitmap supports multiple bits per pixel (because it can be displayed on a color monitor), COLOR specifies the color to be used to draw the line. If COLOR is NIL, the default color, DSPCOLOR, of DISPLAYSTREAM will be used.

5.2.1.2 Relative Drawing

RELDRAWTO draws a line from the current position of the display stream to a point that is displaced DX pixels along the X-axis and DY pixels along the Y-axis. It takes the form:

Function:	RELDRAWTO
# Arguments:	7
Arguments:	1) DX, an X-axis coordinate displacement

Graphics

- 2) DY, a Y-axis coordinate displacement
 - 3) WIDTH, the width of the line
 - 4) OPERATION, the bitblt operation
 - 5) DISPLAYSTREAM, a display stream or bitmap
 - 6) COLOR, the color of the line
 - 7) DASHING, a dashing specification
- The new Y-axis coordinate.

Value:

RELDRAWTO draws a line from the current position in the display stream to a new position which is displaced by DX pixels in the X-direction and DY pixels in the Y-direction. Both DX and DY must be specified and must be non-NIL. Consider the following example, which is depicted in Figure 5-9:

```
<-(SET.DSP.POSITION 100 100 IMS)
(100 . 100)
```

```
<-(RELDRAWTO 100 200 5 'PAINT IMS)
100
```

Graphics

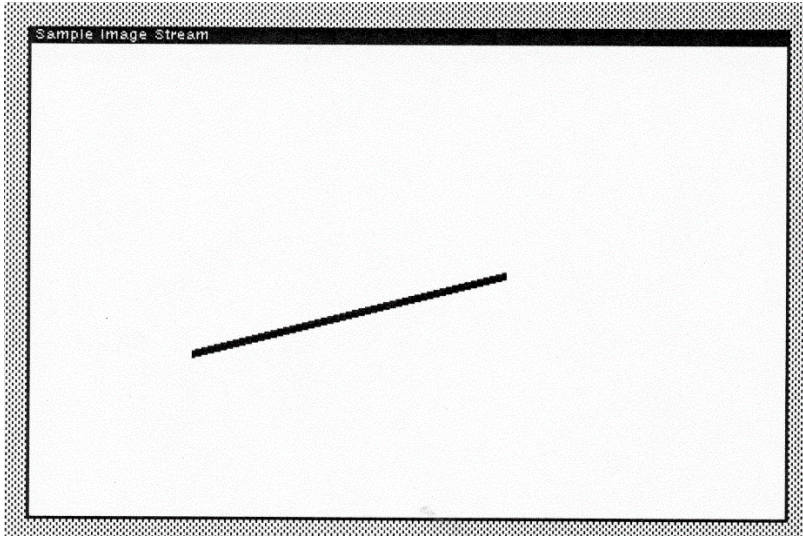


Figure 5-9. Example of RELDRAWTO

5.2.1.3 Absolute Drawing

DRAWLINE draws a line from the point (X1,Y1) to the point (X2,Y2) on the destination display stream. It takes the form:

Function:	DRAWLINE
# Arguments:	9
Arguments:	1) X1, an X-axis coordinate
	2) Y1, a Y-axis coordinate
	3) X2, an X-axis coordinate
	4) Y2, a Y-axis coordinate
	5) WIDTH, the width of the line
	6) OPERATION, the bitblt operation
	7) DISPLAYSTREAM, a display stream or bitmap
	8) COLOR, the color of the line

Graphics

9) DASHING, a dashing specification

Value: A coordinate value.

DRAWLINE draws lines from the point (X1,Y1) to the point (X2,Y2) in the specified display stream. Consider the following example, whose results are depicted in Figure 5-10:

```
<-(DRAWLINE 100 100 100 200 10 'PAINT IMS)
285
<-(DRAWLINE 300 100 300 200 10 'PAINT IMS)
200
<-(DRAWLINE 100 100 300 200 10 'PAINT IMS)
200
<-(DRAWLINE 100 200 300 100 10 'PAINT IMS)
100
```

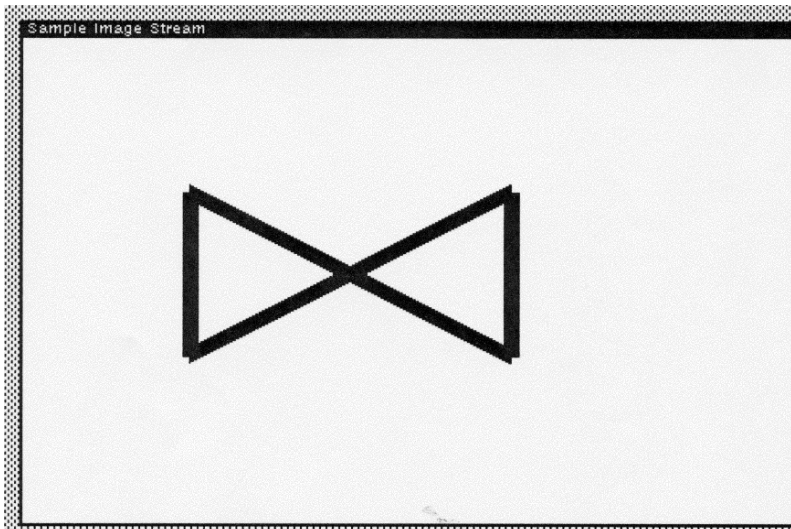


Figure 5-10. Example of DRAWLINE

Graphics

If any of the coordinates are NIL, DRAWLINE returns the error message "NON-NUMERIC ARG".

5.2.1.4 Drawing Between Positions

DRAWBETWEEN draws a line from POS1 to POS2 on the destination display stream. It takes the form:

Function:	DRAWBETWEEN
# Arguments:	7
Arguments:	1) POS1, a position 2) POS2, a position 3) WIDTH, the width of the line 4) OPERATION, the drawing operation 5) DISPLAYSTREAM, a display stream or bit map 6) COLOR, the color of the line 7) DASHING, a dashing specification
Value:	The new Y-axis coordinate.

Consider the following example:

```
<-(SET.DSP.POSITION (POINT 100 100) awindow)
(100 . 100)
```

```
<-(DRAWBETWEEN      (POINT 100 100)
                     (POINT 200 200)
                     100
                     'PAINT
                     awindow)
```

```
200
```


Graphics

which draws a diagonal line between the respective points as depicted in Figure 5-4.

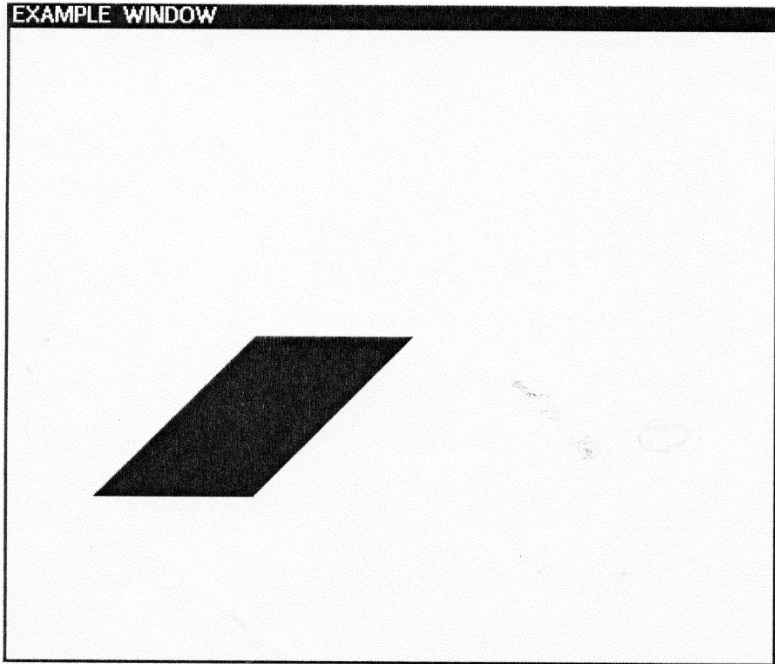


Figure 5-11. An Example of DRAWBETWEEN

If either POS1 or POS2 is NIL, DRAWBETWEEN prints the error message "NON-NUMERIC ARG".

5.2.2 Drawing Curves

A *curve* is described by a set of points to which a spline function will be fitted. **DRAWCURVE** draws a curve in the destination bit map. It takes the form:

Graphics

Function: DRAWCURVE
Arguments: 5
Arguments: 1) KNOTS, a list of trajectory points
2) CLOSED, a flag indicating spline closure
3) BRUSH, the type of brush
4) DASHING, the dashing characteristics
5) DISPLAYSTREAM, a display stream handle
Value: A display stream handle.

Consider the following examples:

```
<-(SETQ KNOTS
      (LIST (POINT 100 100)
            (POINT 125 175)
            (POINT 190 225))
)
((100 . 100) (125 . 175) (190 . 225))

<-(DRAWCURVE KNOTS T NIL NIL IMS)
{STREAM}#64,72470

<-(SETQ KNOTS
      (LIST (POINT 10 10)
            (POINT 200 200)
            (POINT 50 200)
            (POINT 100 50)
            (POINT 200 200))
)
((10 . 10) (200 . 200) (50 . 200) (100 . 50) (200 . 200))

<-(DRAWCURVE KNOTS T NIL NIL IMS)
```

Graphics

{STREAM}#64,72470

These curves are depicted in Figures 5.12 and 5.13 respectively.

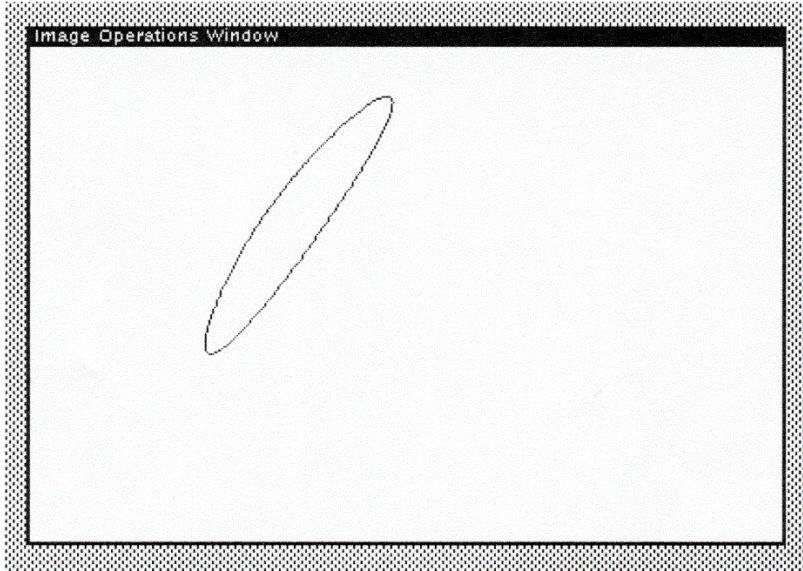


Figure 5-12. A DRAWCURVE Example

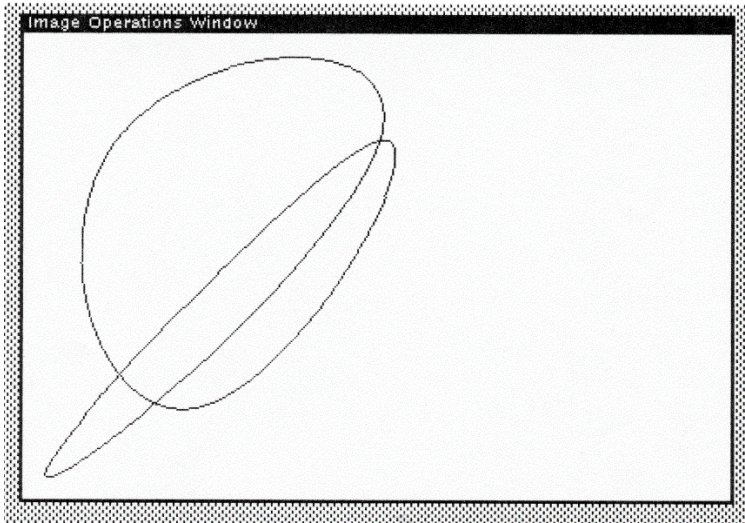


Figure 5-13. Another DRAWCURVE Example

Note that DRAWCURVE processes the points for the curve in sequential order. Thus, lines between points may cross other lines unless the points are sorted. You must ensure that the lines drawn will not cross before you supply the list of points to DRAWCURVE.

5.2.3 Drawing a Gray Box

Interlisp uses gray boxes to outline objects in windows. A system routine, **DRAWGRAYBOX**, allows you to draw a gray box of any proportion in a given window. It takes the following form:

Function:	DRAWGRAYBOX
# Arguments:	6
Arguments:	1) X1, x-coordinate of lower left corner 2) Y1, y-coordinate of lower left corner 3) X2, width in x-direction

Graphics

- 4) Y2, height in y-direction
 - 5) WINDOW/STREAM, a window or display stream handle
 - 6) SHADE, a shade
- Value: A display stream handle.

DRAWGRAYBOX allows you to draw a gray box in the specified window at the location given by (X1,Y1). The dimensions of the box are given by (X2,Y2). The shade of the border of the box is given by SHADE. It may be a number of a bitmap. Consider the following example:

```
<-(DRAWGRAYBOX 100 100 200 200 IOWINDOW  
GRAYSHADE)  
T
```

which is depicted in Figure 5-14.

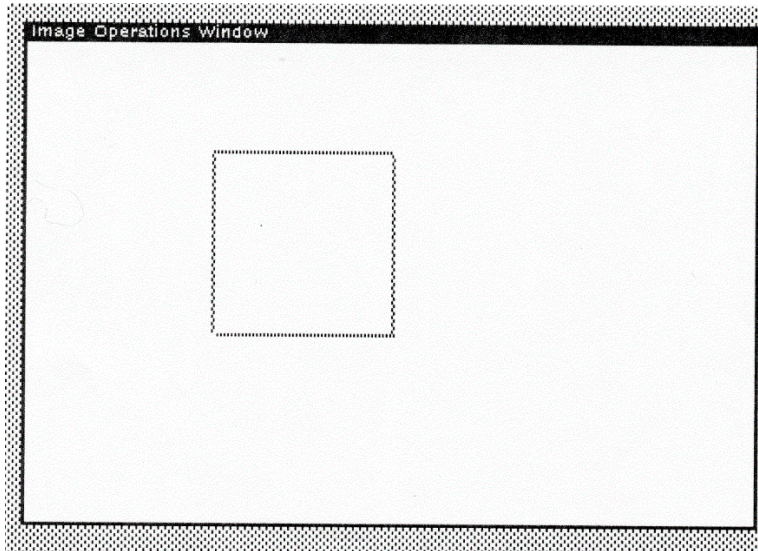


Figure 5-14. Example of DRAWGRAYBOX

5.3 Rectangles

One of the most useful graphics primitives is the ability to draw rectangles. Interlisp does not provide a primitive function for drawing a rectangle (creating a region is a different paradigm). This section will describe a set of functions for creating and manipulating rectangles. A new datatype, RECTANGLE, is created to store information about the rectangle.

A *rectangle* is a four-sided polygon whose opposite sides are both parallel and of equal length. A square is a degenerate rectangle in that all four sides must have the same length. This characteristic allows us to describe a rectangle by exactly two points: the lower left corner and the upper right corner (see figure 5-x).

To describe a rectangle in Medley Interlisp, let us create the record RECTANGLE which consists of two position specifications: the lower left corner, identified by LLCORNER, and the upper right corner, URCORNER. It is defined by:

```
<-(TYPERECORD RECTANGLE (LLCORNER . URCORNER))  
RECTANGLE
```

LLCORNER and URCORNER have a default field type of POINTER which allows them to store position handles.

Note: I do not describe a rectangle as a region because Interlisp currently represents regions as four element lists rather than distinct object. Moreover, regions are abstract objects which are not directly displayed on the screen but are associated with manipulable objects such as windows.

Graphics

5.3.1 Creating a Rectangle

You may create a rectangle by specifying the X and Y coordinates of the lower left and upper right corners. **RECTANGLE** is a function which creates a rectangle:

Function:	RECTANGLE
# Arguments:	4
Arguments:	1) LLX, the lower left X-axis coordinate 2) LLY, the lower left Y-axis coordinate 3) URX, the upper right X-axis coordinate 4) URY, the upper right Y-axis coordinate
Value:	A RECTANGLE datatype handle.

RECTANGLE creates an instance of the datatype RECTANGLE and returns its handle as the value. Let us define RECTANGLE as follows:

```
<-(DEFINEQ (rectangle (llx lly urx ury)
  (create RECTANGLE
    LLCORNER <- (POINT llx lly)
    URCORNER <- (POINT urx ury)
  )
))
(RECTANGLE)
```

Let us create a rectangle whose lower left corner is located at point (100,200) and whose upper right corner is located at point (500,600).

```
<-(SETQ arectangle (RECTANGLE 100 200 500 600))
```

Graphics

```
(RECTANGLE (100 . 200) 500 . 600)
```

We can then access the elements of the rectangle using the record package access mechanisms as follows:

```
<-(fetch LLCORNER of ARECTANGLE)  
(100 . 200)
```

```
<-(fetch XCOORD of (fetch LLCORNER of ARECTANGLE))  
100
```

and so on.

5.3.1.1 Creating a Square

As we mentioned above, a square is a degenerate form of a rectangle which has all sides of equal length. We can use RECTANGLE to create a square, but we would like to ensure that the sides are of equal length. SQUARE takes a point (X,Y) and the length of the sides as its arguments, creates a square rectangle, and returns the rectangle handle. It takes the form:

Function:	SQUARE
# Arguments:	3
Arguments:	1) LLX, the lower left X-axis coordinate 2) LLY, the lower left Y-axis coordinate 3) LENGTH, the length of a side
Value:	A rectangle handle.

Let us define SQUARE as follows:

```
<-(DEFINEQ (square (llx lly length)  
(rectangle llx lly
```


Graphics

```
(iplus llx length)
(iplus lly length)
)
))
(SQUARE)
```

Thus, if we wanted to create a square whose origin is (200,200) and whose sides have length 300, we would execute the following expression:

```
<-(SQUARE 200 200 300)
(RECTANGLE (200 . 200) 500 . 500)
```

```
<-(SQUARE 200 200 -250)
(RECTANGLE (200 . 200) -50 . -50)
```

5.3.1.2 Accessing Rectangle Parameters

Because a rectangle is a user-defined datatype, you may access the parameters describing the rectangle using the record package access functions. However, expressions involving these access functions may be quite lengthy, so we define two utility functions to provide this information: **LOWERLEFT** and **UPPERRIGHT**, which take the form:

Function:	LOWERLEFT UPPERRIGHT
# Arguments:	1
Arguments:	1) RECTANGLE, a rectangle handle
Value:	The respective point (X,Y) specified by the function.

Let us define these functions as follows:

```
<-(DEFINEQ (lowerleft (arectangle)
```

Graphics

```
(IF (TYPE? RECTANGLE arectangle)
  THEN
  (fetch LLCORNER of arectangle))
))
(LOWERLEFT)

<-(DEFINEQ (upperright (arectangle)
  (IF (TYPE? RECTANGLE arectangle)
    THEN
    (fetch URCORNER of arectangle)))
))
(UPPERRIGHT)
```

Consider the following example:

```
<-(SETQ SQ1 (SQUARE 100 100 150))
(RECTANGLE (100 . 100) 250 . 250)

<-(LOWERLEFT SQ1)
(100 . 100)

<-(UPPERRIGHT SQ1)
(250 . 250)
```

5.3.1.3 Displaying a Rectangle

You may display a rectangle within a window using the function **DISPLAY.RECTANGLE**, which takes the form:

Function:	DISPLAY.RECTANGLE
# Arguments:	2
Arguments:	1) WINDOW, a window handle 2) RECTANGLE, a rectangle handle
Value:	The X-coordinate.

Graphics

We might define DISPLAY.RECTANGLE as follows:

```
<-(DEFINEQ (display.rectangle (window rectangle)
  (IF (TYPE? rectangle RECTANGLE)
    THEN
      (DRAWBETWEEN (LOWERLEFT rectangle)
        (POINT (fetch XCOORD of (LOWERLEFT rectangle))
              (fetch YCOORD of (UPPERRIGHT
                rectangle)))
          1
          PAINT
          window)
      (DRAWBETWEEN (LOWERLEFT rectangle)
        (POINT (fetch XCOORD of (LOWERLEFT rectangle))
              (fetch YCOORD of (UPPERRIGHT rectangle)))
          1
          'PAINT
          window)
      (DRAWBETWEEN (UPPERRIGHT rectangle)
        (POINT (fetch YCOORD of (UPPERRIGHT rectangle))
              (fetch XCOORD of (LOWERLEFT rectangle)))
          1
          'PAINT
          window)
      (DRAWBETWEEN (UPPERRIGHT rectangle)
        (POINT (fetch YCOORD of (LOWERLEFT rectangle))
              (fetch XCOORD of (UPPERRIGHT rectangle)))
          1
          'PAINT
          window)
    )
  ))
(DISPLAY.RECTANGLE)
```

Consider the following example which is depicted in Figure 5-15:

Graphics

```
<-SQ1  
(RECTANGLE (100 . 100) 250 . 250)  
  
<-(DISPLAY.RECTANGLE IOWINDOW SQ1)  
100
```

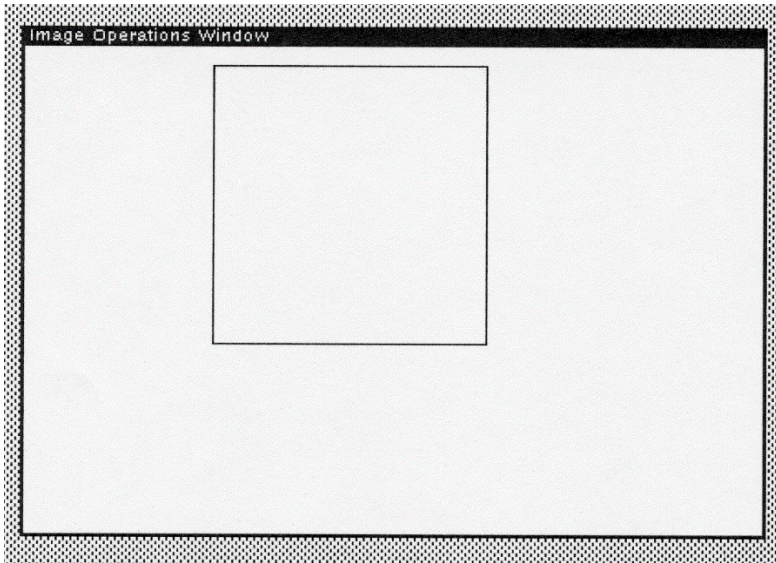


Figure 5-15. RECTANGLE Example

5.3.2 Rectangle Functions

A number of commonly used functions may be defined for rectangles. These are described and defined in the following sections.

5.3.2.1 Height and Width

Graphics

The height and width of a rectangle are used in many applications. Let us define two functions - **HEIGHT** and **WIDTH** - which calculate these values for an arbitrary rectangle. They take the form:

Function:	HEIGHT WIDTH
# Arguments:	
Arguments:	1) RECTANGLE, a rectangle handle
Value:	The respective measurement specified by the function.

We can define HEIGHT and WIDTH as follows:

```
<-(DEFINEQ (height (rectangle)
  (IDIFFERENCE
    (fetch YCOORD of (UPPERRIGHT rectangle))
    (fetch YCOORD of (LOWERLEFT rectangle))))
(HEIGHT))
```

```
<-(DEFINEQ (width (rectangle)
  (IDIFFERENCE
    (fetch XCOORD of (UPPERRIGHT rectangle))
    (fetch XCOORD of (LOWERLEFT rectangle))
  ))
(WIDTH))
```

Consider the following examples:

```
<-(WIDTH arectangle)
300
```

```
<-(HEIGHT arectangle)
300
```

Graphics

5.3.2.2 Finding The Rectangle's Center

You may find the rectangle's center by executing **CENTER**, which takes the form:

Function:	CENTER
# Arguments:	1
Arguments:	1) RECTANGLE, a rectangle handle
Value:	A position.

CENTER returns the position whose components specify the center point of the rectangle. Note that CENTER uses integer arithmetic, and so, the center may be a pixel off in either direction if the length of either of the sides of the rectangle is an odd number. We may define CENTER as follows:

```
<-(DEFINEQ (center (arectangle)
  (IF (TYPE? RECTANGLE arectangle)
    THEN
      (POINT
        (IQUOTIENT
          (IPLUS (fetch XCOORD of
            (LOWERLEFT arectangle))
            (fetch XCOORD of
            (UPPERRIGHT arectangle)))
          2)
        (IQUOTIENT
          (IPLUS (fetch YCOORD of
            (LOWERLEFT arectangle))
            (fetch YCOORD of
            (UPPERRIGHT arectangle)))
          2)
      )
    )
  )
))
```

Graphics

Consider the following example:

```
<-SQ1  
(RECTANGLE (100 . 100) 250 . 250)
```

```
<-(CENTER SQ1)  
(175 . 175)
```

5.3.3 Rectangle Manipulation Functions

This section describes several function for manipulating rectangles that I have created as utility functions.

5.3.3.1 Expanding a Rectangle

You may expand the dimensions of a rectangle by moving its upper right corner and lower left corner in the X or Y directions. To do so, you may use the function **EXPAND.RECTANGLE**, which takes the form:

Function:	EXPAND.RECTANGLE
# Arguments:	3
Arguments:	1) ARECTANGLE, a rectangle handle 2) DELTAX, an X-axis increment 3) DELTAY, a Y-axis increment
Value:	A new rectangle.

We may define EXPAND.RECTANGLE as follows:

```
<-(DEFINEQ (expand.rectangle (arectangle deltax deltax)  
  (RECTANGLE  
    (IDIFFERENCE  
      (fetch XCOORD of (LOWERLEFT arectangle))  
      deltax)
```

Graphics

```
(IDIFFERENCE
  (fetch YCOORD of (LOWERLEFT arectangle))
  deltax)
(IPLUS
  (fetch XCOORD of (UPPERRIGHT arectangle))
  deltax)
(IPLUS
  (fetch YCOORD of (UPPERRIGHT arectangle))
  deltax)
)
))
(EXPAND.RECTANGLE)
```

Consider the following example:

```
<-SQ1
(RECTANGLE (100 . 100) 250 . 250)

<-(EXPAND.RECTANGLE SQ1 100 100)
(RECTANGLE (0 . 0) 350 . 350)
```

5.3.3.2 Finding a Rectangle's Area

You may compute the area of a rectangle using the function **AREA.OF.RECTANGLE**, which takes the form:

Function:	AREA.OF.RECTANGLE
# Arguments:	1
Arguments:	1) ARECTANGLE, a rectangle handle
Value:	An integer.

We may define AREA.OF.RECTANGLE as follows:

```
<-(DEFINEQ (AREA.OF.RECTANGLE (arectangle)
  (IF (TYPE? arectangle 'RECTANGLE)
```


Graphics

```
    THEN
    (TIMES (WIDTH arectangle) (HEIGHT arectangle)))
))
(AREA.OF.RECTANGLE)
```

Consider the following example:

```
<-(AREA.OF.RECTANGLE arectangle)
20000
```

5.4 Closed Polygons

Interlisp provides two functions for drawing closed curved polygons, e.g., circles and ellipses. We provide you with the definitions for drawing two additional types of closed polygons: triangles and hexagons.

5.4.1 Drawing Circles

You may draw a circle in a window using the function **DRAWCIRCLE**, which takes the form:

Function:	DRAWCIRCLE
# Arguments:	6
Arguments:	1) X, the x-coordinate of the center 2) Y, the y-coordinate of the center 3) RADIUS, the radius of the circle 4) BRUSH, a brush handle 5) DASHING, a dashing specification 6) DISPLAYSTREAM, a display stream handle
Value:	NIL.

DRAWCIRCLE draws a circle of size RADIUS centered about the point (X,Y) in the destination bit map of the specified display

Graphics

stream. The current position of DISPLAYSTREAM will be the position (X,Y). Consider the following example:

```
<-(DRAWCIRCLE 100 100 100 NIL NIL awindow)  
NIL
```

which draws the following figure:

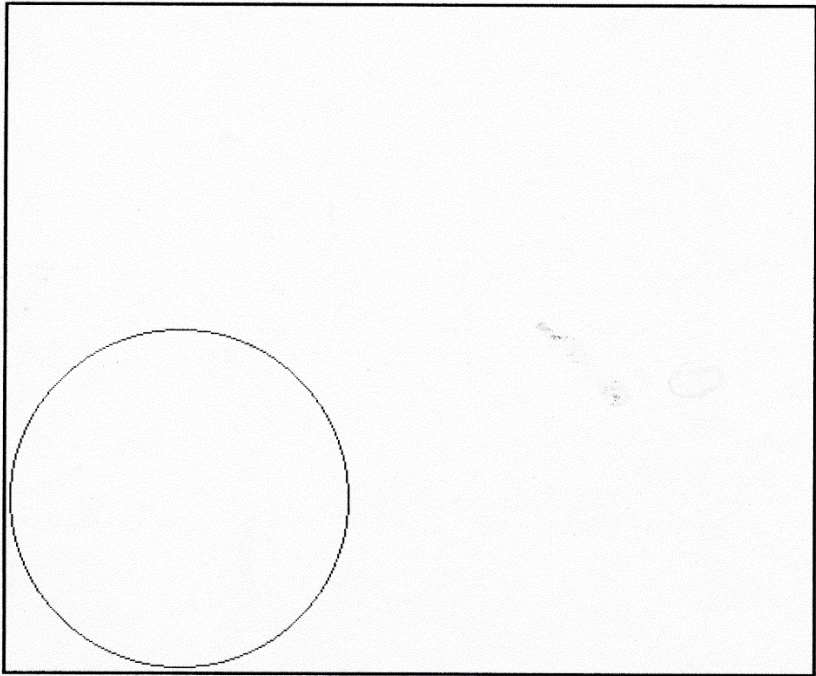


Figure 5-16. DRAWCIRCLE Example

In this example, the brush default of 1 has been used and no dashing has been specified.

Graphics

5.4.2 Drawing Ellipses

You may draw an ellipse (a lop-sided circle) using the function **DRAWELLIPSE**, which takes the form:

Function:	DRAWELLIPSE
# Arguments:	8
Arguments:	1) X, the x-coordinate of the center 2) Y, the y-coordinate of the center 3) SEMIMINORRADIUS, the minor radius 4) SEMIMAJORRADIUS, the major radius 5) ORIENTATION, the angle of the major axis 6) BRUSH, a brush handle 7) DASHING, a dashing specification 8) DISPLAYSTREAM, a display stream handle
Value:	NIL.

DRAWELLIPSE draws an ellipse about the position (X,Y) in the destination bit map of the given display stream with a minor radius of **SEMIMINORRADIUS** and a major radius of **SEMIMAJORRADIUS**. The ellipse is oriented with the major axis having an angle of **ORIENTATION** degrees. The orientation is positive in a counterclockwise direction. The current position of **DISPLAYSTREAM** is the position (X,Y).

Consider the following example which is depicted in Figure 5-17:

```
<-(DRAWELLIPSE 100 100 50 100 45 3 NIL awindow)
NIL
```

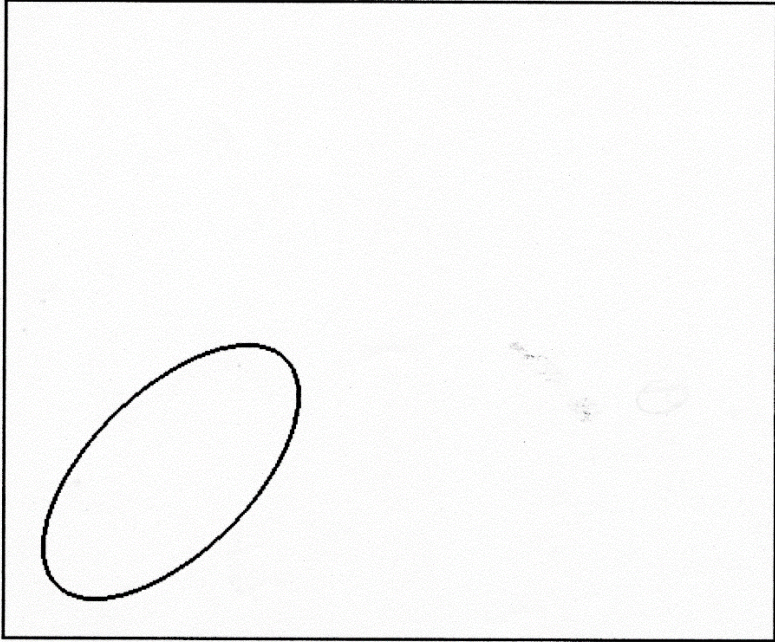


Figure 5-17. DRAWELLIPSE Example

5.5 Filling Objects with Texture

The previous sections have described functions that allow you to draw various types of polygons on the screen. The ability to fill those polygons goes a long way towards enhancing the presentation of information that is displayed on the screen. Interlisp provides two functions for filling arbitrary polygons and circles on the screen. You should also consult DSPFILL for filling arbitrary regions.

5.5.1 Filling Polygons

Graphics

You may fill an arbitrary polygon described by its vertices using the function **FILLPOLYGON**, which takes the form:

Function:	FILLPOLYGON
# Arguments:	3
Arguments:	1) POINTS, a list of points representing the vertices of the polygon 2) TEXTURE, a texture handle 3) STREAM, a display stream handle
Value:	NIL.

FILLPOLYGON fills in the polygon described by POINTS with TEXTURE. POINTS is a list of positions which determine the vertices of a closed polygon. The positions occurring in POINTS are assumed local to STREAM.

The entries in POINTS may be lists which describe separate polygons to be filled. Thus, you can fill several polygons with one list or you can fill a set of polygons which might be nested inside one another.

When filling a polygon, consideration must be given to the case where two polygon sides intersect. FILLPOLYGON uses an "odd" fill rule which means that intersecting polygons define areas like a checkerboard which are filled or not filled. Consider the example:

```
<-(FILLPOLYGON '((125 . 125)
                 (150 . 200)
                 (175 . 125)
                 (125 . 175)
                 (175 . 175))
                GRAYSHADE
                WINDOW)
NIL
```

Graphics

produces a display as depicted in Figure 5-18.

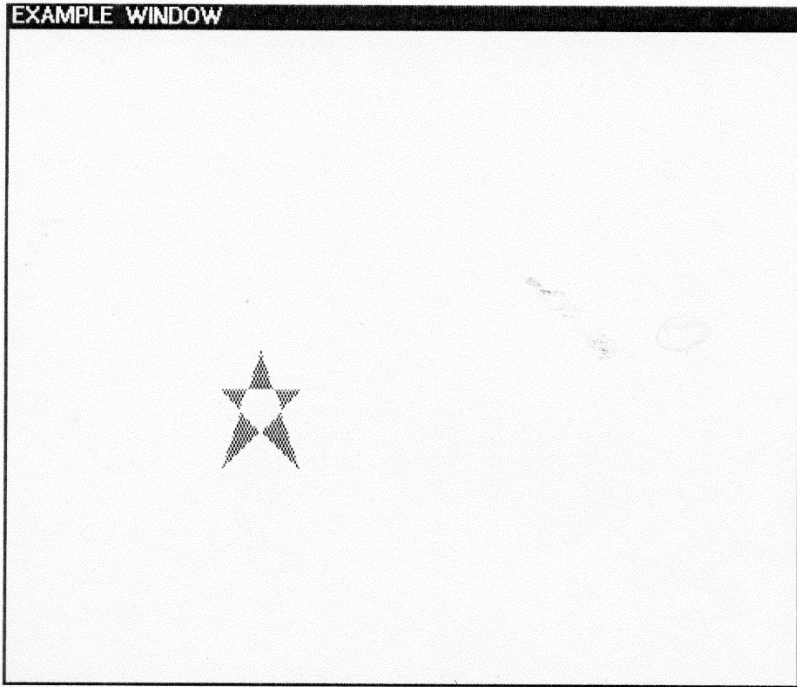


Figure 5-18. An Example Using FILLPOLYGON

If POINTS specifies multiple polygons, the fill rule described above can be used to place "holes" in filled polygons. Consider the following example:

```
<-(FILLPOLYGON  '(((110 . 110)
                  (150 . 200)
                  (190 . 110))
                  ((135 . 125)
                  (160 . 125)
                  (160 . 150)
```

Graphics

```
(135 . 150)))  
GRAYSHADE  
WINDOW)
```

NIL

produces a square hole in a triangular region in WINDOW.

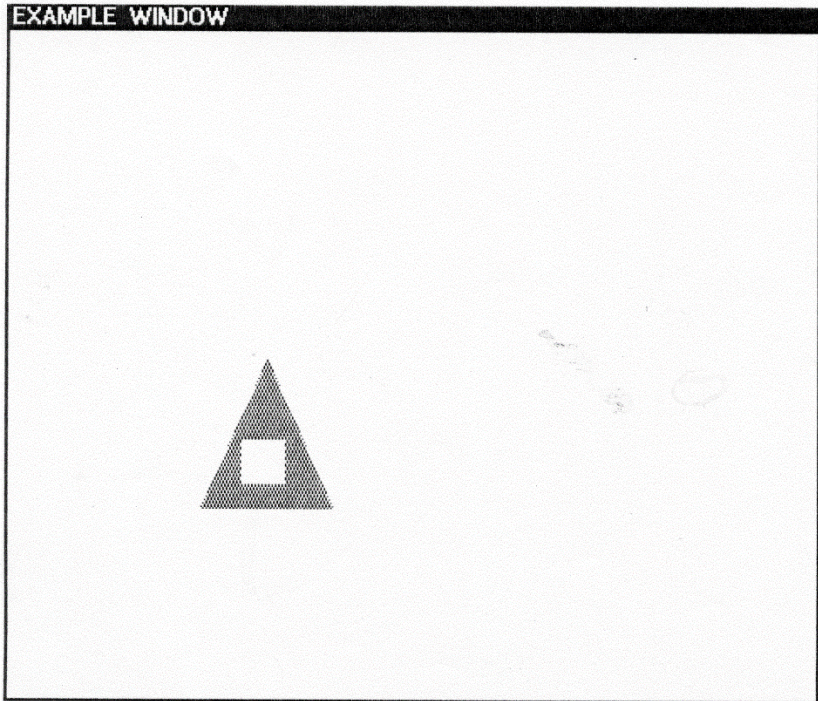


Figure 5-19. Another Example of FILLPOLYGON

FILLPOLYGON uses the "replace" operation to fill areas with texture. Any areas not filled are not changed. If there are "holes" in the filled polygon, they can be used to produce a "windowing" effect.

Graphics

5.5.2 Filling a Circle

You may fill a circle with a texture using the function **FILLCIRCLE**, which takes the form:

Function:	FILLCIRCLE
# Arguments:	5
Arguments:	1) CENTERX, X-coordinate of center 2) CENTERY, Y-coordinate of center 3) RADIUS, radius of circle 4) TEXTURE, a texture handle 5) STREAM, a display stream handle
Value:	NIL.

FILLCIRCLE fills in a circular region about the point (CENTERX, CENTERY) with the given texture to the specified radius. Consider the following example which fills a circle with the checkerboard texture:

```
<-(FILLCIRCLE 150 150 50 CHECKBRUSH IOWINDOW)  
NIL
```


Graphics

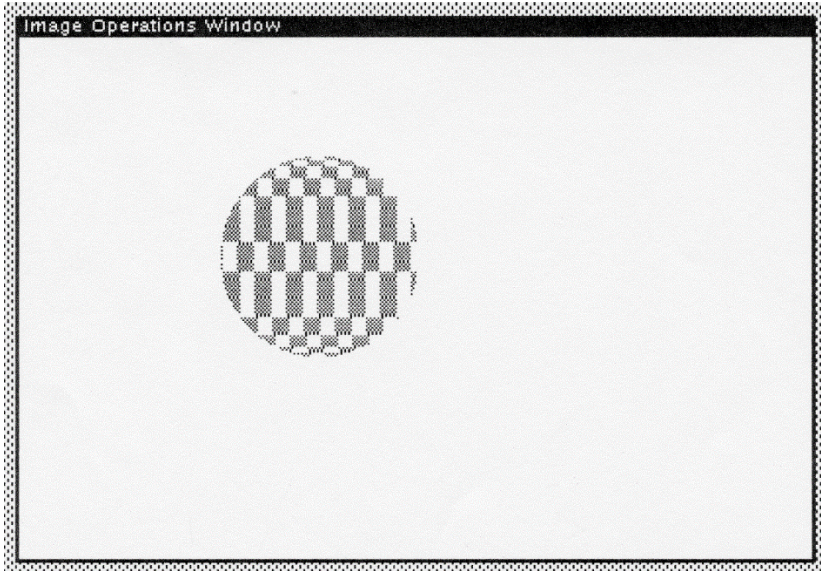


Figure 5-20. FILLCIRCLE Example

6. Process Management

Multiprocessing (actually, multitasking) is a recent innovation in Interlisp, although other Lisps (such as FranzLisp) have supported some type of mechanism either directly or through access to the operating system mechanisms. Interlisp process management allows you to initiate multiple Lisp processes which may operate asynchronously, but which may be synchronized through an interprocess communication mechanism. Each Interlisp process executes in its own stack, but all Interlisp processes share the same global address space provided by the Interlisp virtual memory.

Interlisp assumes a benign environment, i.e., all processes cooperate together. Thus, process switching occurs on a voluntary basis, rather than being forced by the underlying virtual machine. There is no preemption mechanism nor is there a provision for guaranteeing service. A process may run for as long as it needs to.

The process mechanism is an integral component of the Interlisp programming environment. Most of the standard system facilities (such as network operations) require that the process mechanism be active. I caution you against disabling it unless an exceptional condition occurs (such as catastrophic failure of an applications program where essential data must be saved).

CAVEAT: This facility was developed for the Xerox D Machines. It has not yet been evaluated to run on the modern machines to which Medley Interlisp has been ported.

6.1 Process Concepts

A *process* is a locus of control. Each process is defined by a *process handle*, which takes the form:

PROCESS MANAGEMENT

{PROCESS}#<address>

where <address> is a memory location.

6.1.1 The Structure of a Process

A process is described by a *process object*, which is a data structure maintained in memory by Interlisp. The process object has the structure presented in Table 6-1.

Table 6-1. Process Structure

Field	Type	Description
PROCRESTARTFORM	POINTER	An expression that is executed when a process restarted.
PROCDRIBBLEOFD	POINTER	A flag
PROCTTYEXITFN	POINTER	A function applied to process name when it ceases to be the TTY process.
PROCTTYENTRYFN	POINTER	A function applied to process name when it becomes the TTY process.
PROCEVAPPLYRESULT	POINTER	Storage for PROCESS.EVAL and PROCESS.APPLY when WAITFORRESULT is true.
PROCOWNEDLOCKS	POINTER	A pointer to the first monitor lock owned by the process.
PROCBEFOREEXIT	POINTER	A function invoked before the process terminates; may be used to set up return values from the process.

PROCESS MANAGEMENT

PROCAFTEREXIT	POINTER	A function invoked when restarting Interlisp after a SYSOUT, etc.
PROCEVENTLINK	POINTER	A pointer to the event for the process.
PROCUSERDATA	POINTER	Storage for user properties for the process.
PROCREMOTEINFO	POINTER	A flag used in network communications.
PROCTYPEAHEAD	POINTER	Address of buffer which contains typeahead for this process.
PROCINFOHOOK	POINTER	Optional user function that displays information about the process.
PROCRESETVARLIST	POINTER	Bindings for RESETVARLIST in the process.
PROCMAILBOX	POINTER	A queue of messages.
PROCFINISHEVENT	POINTER	An optional event to be notified when the process finishes.
PROCRESULT	POINTER	The value to be returned if the process finishes normally.
PROCFINISHED	POINTER	Value indicating how the process finished: NORMAL, DELETED, or ERROR.
PROCWINDOW	POINTER	The window handle of a window associated with this process, if any.
RESTARTABLE	POINTER	If T, autorestart on error; HARDRESET means restart only on a hard reset.
PROCFORM	POINTER	The form to EVAL to initiate the process.
PROCEVENTORLOCK	POINTER	An event or monitor lock that the process is waiting for.

PROCESS MANAGEMENT

WAKEREASON	POINTER	The reason the process is being run.
PROCTIMERBOX	POINTER	A scratch box to use for PROCWAKEUPTIMER when the user does not provide one.
PROCTIMERLINK	POINTER	Links the process to the
PROCWAKEUPTIMER	POINTER	The time this process last went to sleep.
PROCNEVERSTARTED	FLAG	T, if the process has never been started.
PROCSYSTEMP	FLAG	A flag.
PROCDELETED	FLAG	A flag indicating the process has been deleted; cleanup occurs later.
PROCBEINGDELETED	FLAG	T, if the process was deleted but hasn't yet been removed from \PROCESSES.
PROCTIMERSET	FLAG	T, if PROCWAKEUPTIMER has an interesting value.
NEXTPROHANDLE	POINTER	Pointer to the next process handle.
PROCQUEUE	POINTER	Pointer to the queue of processes of the same priority.
PROC PRIORITY	BYTE	Priority level of the process; currently 0-4.
PROCNAME	POINTER	The name of the process.
PROCSTATUS	BYTE	The process status.
PROCFX	WORD	The stack pointer for this process when it is asleep.
PROCFX0	WORD	Is \STACKHI to make it look like a STACKP.

This data structure may be examined by the Inspector as follows:

PROCESS MANAGEMENT

<-(INSPECT (DF PRINT.FILE))

which yields the following structure (with some fields filled in explicitly for DEdit).

PROCSTARTFORM	NIL
PROCDRIBBLEOFD	NIL
PROCTTYEXITFN	DEDITTABSOFF
PROCTTYENTRYFN	DEDITTABSON
PROCEAPPLYRESULT	NIL
PROCOWNEDLOCKS	NIL
PROCBFOREEXIT	NIL
PROCAFTEREXIT	NIL
PROCEVENTLINK	{PROCESS}#65,24700
PROCUSERDATA	(TABACTION (IGNORE . IGNORE))
PROCREMOTEINFO	NIL
PROCTYPEAHEAD	NIL
PROCINFOHOOK	NIL
PROCRESETVARLIST	((\DEDITALLOWSELS) (\DEDITUSER NIL)) (SETCURSOR ({BITMAP}#57,134520 0 . 15)) NIL) (UNEDITW ({WINDOW}#64,152064)) NIL) (\DEDITSELECTIONS) (EDITF2 PRINT.FILE NIL FNS NIL (LAMBDA (FILE) (PROG (OLD-RPARS) (SETQ OLD-COLUMN FIRSTCOL) (SETQ OLD-RPARS #RPARS)

PROCESS MANAGEMENT

```
(SETQ #RPARS NIL)
(SETQ 4045STREAM
  (OPENIMAGESTREAM ...))
(LINELENGTH 70 4045STREAM)
(OPENSTREAM FILE 'INPUT)
(EVAL (LIST 'SEE* FILE)
  4045STREAM)
(CLOSEF 4045STREAM)
(SETQ #RPARS OLD-RPARS))))
NIL))
PROCMAILBOX      NIL
PROCFINISHEVENT  NIL
PROCRESULT       NIL
PROCFINISHED     NIL
PROCWINDOW       NIL
RESTARTABLE      NIL
PROCFORM         (EDITDEF (QUOTE PRINT.FILE
  (QUOTE FNS)))
PROCEVENTORLOCK  {EVENT}#71,17710
WAKEREASON       NIL
PROCTIMERBOX     0
PROCTIMERLINK    NIL
PROCWAKEUPTIMER  NIL
PROCNEVERSTARTED  NIL
PROCSYSTEMP     NIL
PROCDELETED      NIL
PROCBEINGDELETED  NIL
PROCTIMERSET     NIL
NEXTPROCHANDLE   NIL
PROCQUEUE        {PROCESSQUEUE}#71,20770
PROCPRRIORITY    2
PROCNAME         EDITDEF
PROCSTATUS       0
PROCFX          27224
PROCFX0         1
```

PROCESS MANAGEMENT

Each process may also be described by certain properties which affect its behavior.

6.1.2 The TTY Process

The TTY Process is a global role that is created when Interlisp is initialized. It is assumed to be the only process which will receive input from the keyboard. For any other process to receive input, it must explicitly assume the role of the TTY Process through a function call. Initially, this role is assigned to the process associated with the Interlisp Executive Window which processes top-level expressions typed by the user.

When the TTY Process role is switched to another process, any input that was typed ahead is saved in the current processes' object. Thus, characters typed at the keyboard are always sent to the process which assumed the role of the TTY Process at the time the characters were typed.

A process may either assign the TTY Process role to itself or to another process. Typically, the latter case occurs when an executive process within your program determines which process should perform the next step of the computation or it is expecting a certain type of input to be entered by the user.

From the user's viewpoint, you want to designate which process will receive input by clicking the mouse in the window associated with the process. Thus, any process which wants to receive keyboard input should place its process handle in the PROCESS property of its window(s). When (TTYDISPLAYSTREAM) switches to a new window, it automatically performs this action.

6.1.2.1 Switching the TTY Process Role

PROCESS MANAGEMENT

Any process can assume the role of the TTY Process by executing the function **TTY.PROCESS**, which takes the form:

Function:	TTY.PROCESS
# Arguments:	1
Arguments:	1) PROC, a process handle
Value:	The handle of the current process.

If PROC is NIL, the current process assumes the role of the TTY Process. The handle of the current TTY Process is returned as the value. Assume the TTY process currently resides in the Interlisp Executive Window.

```
<-(TTY.PROCESS)
{PROCESS}#71,22400
```

Typically, when the handle of a process is returned, you will cache it in a variable or property so that you can reinstate that process at a later time.

If PROC is non-NIL, that process is assigned the role of the TTY Process. This permits one process to assume control of the computation, but allows another to receive and process input designated for the program.

Let MYPROCESS contain the process handle of a TEdit process. Consider the following example:

```
<-MYPROCESS
{PROCESS}#71,110500

<-(TTY.PROCESS MYPROCESS)
{PROCESS}#71,22400
```

PROCESS MANAGEMENT

When PROC has the value T, the TTY Process role is assigned to the Executive Process, which handles top-level interactions with the user.

Notifying a Process of its TTY Process Role

Since a process can be assigned the TTY Process role by another process, there are cases when the process wants to be notified that it has been assigned this role. The process property TTYENTRYFN should contain a function which performs the necessary housekeeping chores upon assumption of the TTY Process role.

Giving up the TTY Process Role

When a process gives up the TTY Process role or another process seizes it, the process losing that role may want to be notified. The process property TTYEXITFN should contain a function which performs the necessary housekeeping chores upon losing or surrendering the TTY Process role.

TTY Process Role Assignment Procedure

In summary, the algorithm for TTY.PROCESS operates as follows:

1. The former process having the TTY Process role has the function assigned to its TTYEXITFN process property executed with two arguments: (OLDTTYPROCESS NEWTTYPROCESS).
2. The new process is assigned the TTY Process role.
3. The new process having the TTY Process role has the function associated with its TTYENTRYFN process property executed.

6.1.2.2 Testing for the TTY Process Role

You may test if a process has the TTY Process role assigned to it using **TTY.PROCESSP**, which takes the form:

PROCESS MANAGEMENT

Function: TTY.PROCESSP
Arguments: 1
Arguments: 1) PROC, a process handle

Value: T or NIL.

If PROC is NIL, TTY.PROCESSP always tests the current process. Thus, the following expression:

```
<-(TTY.PROCESSP)  
T
```

```
<-(TTY.PROCESSP MYPROCESS)  
NIL
```

is true if the current process has the role of the TTY Process assigned to it. Of course, its always true at the top level.

6.1.2.3 Waiting for the TTY Process

You may wait for a process to assume the TTY Process role by executing the function **WAIT.FOR.TTY**, which takes the form:

Function: WAIT.FOR.TTY
Arguments: 2
Arguments: 1) MSECS, the number of milliseconds to wait
2) NEEDWINDOW, a flag

Value: T

WAIT.FOR.TTY efficiently waits for the current process to assume the role of the TTY Process. MSECS is the number of milliseconds to wait before timing out.

PROCESS MANAGEMENT

If `WAIT.FOR.TTY` times out, it returns `NIL`. However, if the current process assumes the role of the TTY Process within `MSECS`, `WAIT.FOR.TTY` returns `T`.

If `MSECS` is `NIL`, `WAIT.FOR.TTY` does not timeout; the current process waits (possibly) forever.

If `NEEDWINDOW` is non-`NIL`, `WAIT.FOR.TTY` opens a window for the current process if one is not already open (e.g., `PROCWINDOW` is `NIL`). This window is opened using `TTYIN`.

`WAIT.FOR.TTY` will spawn a new mouse process (see Section 6.1.4, II) if it is called under the mouse process (see `SPAWN.MOUSE`).

`WAIT.FOR.TTY` is usually used internally by Interlisp processes that need to read from the terminal.

`WAIT.FOR.TTY` is an essential function since the keyboard is a scarce resource. Access to it must be controlled among the competing processes when a user can type commands to several different processes as opposed to typing to one user interface process which then distributes the data to the requesting processes.

6.1.2.4 Clicking the Mouse in a Window

Each window has the property `WINDOWENTRYFN`, which controls whether or not the TTY Process role is assigned to the process associated with the window.

The mouse handler, before invoking the function assigned as the value of the window property `BUTTONEVENTFN`, determines whether a mouse button has been pressed in a window which is associated with a process. If the process is not the TTY Process (e.g., has not assumed that role), it invokes the window's

WINDOWENTRYFN to determine what to do. The default value of the WINDOWENTRYFN property for all windows is the function GIVE.TTY.PROCESS.

The WINDOWENTRYFN may establish the mouse environment before processing the button event. For example, it can move the cursor to a specific object or display a menu that the user must choose from.

6.1.2.5 Giving the TTY Process Role to a Window Process

You may give the TTY Process role to a process associated with a window by executing the function **GIVE.TTY.PROCESS**, which takes the form:

Function:	GIVE.TTY.PROCESS
# Arguments:	1
Arguments:	1) WINDOW, a window handle
Value:	The window handle.

If WINDOW has a non-NIL value for the PROCESS property, GIVE.TTY.PROCESS executes the following expression:

```
(TTY.PROCESS (WINDOWPROP WINDOW 'PROCESS))
```

and then invokes the function associated with the WINDOW's BUTTONEVENTFN property. Alternatively, if the right button was pressed, it will execute the function associated with the RIGHTBUTTONFN property.

Ownership of Windows

If you follow a hierarchical decomposition methodology in the construction of your program, most processes will own a disjoint set of windows. In some cases, two or more process may own the same

PROCESS MANAGEMENT

set of windows. Unless the processes observe a strict cooperative protocol, they may confuse each other with respect to the use of the windows and the assignment of the TTY Process role. It is your responsibility to provide mechanisms to avoid confusion (possibly using events or monitors).

A global window, like the Prompt Window, is used both by user processes and system processes. You should not have your processes reading from the Prompt Window.

Thus, each process is able to print to its own primary output or terminal and read from its own primary input or the terminal without interfering with other processes.

Each process is initialized with its primary and terminal input and output streams set to a default. When the process attempts to read or write to the terminal, a TTY window will be automatically created for the process. The region where the TTY window will be created is given by the system variable `DEFAULTTTYREGION`.

A process may call `TTYDISPLAYSTREAM` at any time to acquire a TTY window explicitly. `TTYDISPLAYSTREAM` sets both the terminal and primary input and output streams to the chosen window.

6.1.2.6 Determining if a Process has a TTY Window

A process may determine if it has a TTY window using the function `HASTTYWINDOWP`, which takes the form:

Function:	<code>HASTTYWINDOWP</code>
# Arguments:	1
Arguments:	1) <code>PROCESS</code> , a process handle

PROCESS MANAGEMENT

Value: T, if the process has a
TTY window.

HASTTYWINDOWP determines if the given process has a TTY window. If so, it returns T; otherwise, NIL. Consider the following example:

```
<-(HASTTYWINDOWP)  
T
```

because the Exec window certainly has the TTY window when I can type into it.

6.1.3 Handling the Mouse

The window mouse handler runs as an independent process. When **BUTTONEVENTFN** of a window is being executed as a result of clicking a mouse button in a window, the mouse handler is not available to perform other functions or window operations. This will cause two problems:

1. Long computations initiated by the **BUTTONEVENTFN** deprive you of the use of the mouse for concurrent activities.
2. Functions executed as parts of **BUTTONEVENTFN**s cannot rely on other **BUTTONEVENTFN**s running concurrently; thus, some function sequences will run differently when executed under the control of the mouse process.

Two functions allow you additional control over mouse processes as described in the following sections.

6.1.3.1 Spawning Mouse Processes

You may spawn additional mouse processes using the function **SPAWN.MOUSE**, which takes the form:

PROCESS MANAGEMENT

Function:	SPAWN.MOUSE
# Arguments:	0
Arguments:	N/A
Value:	T

SPAWN.MOUSE creates another mouse process which allows the mouse to be used concurrently by multiple mouse processes. Consider the following example:

```
<-(SPAWN.MOUSE)  
T
```

This function is largely intended for use inside a process where you want to allow independent mouse events in different windows. Each time you spawn a mouse process, the mouse process is attached to a window. The following function handles the button events in that window.

6.1.3.2 Allowing Mouse Button Events

When you enter a BUTTONEVENTFN as a result, scrolling is specified as the value of the window property SCROLLFN. If this value is NIL, the window is **not scrollable**. The function assigned to this property takes four arguments:

1. The window being scrolled.
2. The distance to scroll in the horizontal direction.
3. The distance to scroll in the vertical direction.
4. A flag which is T if a mouse button is held down while in the scrolling region.

For arguments (2) and (3), a positive number indicates either right or up, while a negative number indicates either left worrying about "flooding" the system with numerous mouse processes. The window

PROCESS MANAGEMENT

mouse handler arranges to terminate itself if it returns from a `BUTTONEVENTFN` and detects another mouse process is operating.

6.1.4 Handling Interrupts

When you "strike" a keyboard interrupt character, any process could be running. A decision must be made about how to handle the interrupt. Most interrupts will be handled by the current TTY Process (e.g., the process having the TTY Process role). Certain interrupts are handled in a special fashion as described in the following paragraphs.

6.1.4.1 Handling Reset/Error Interrupts

Two interrupts result in the termination of the computation: `RESET` (initially, `<CTRL-D>`) and `ERROR` (initially, `<CTRL-E>`).

These interrupts are handled in the mouse process, if the mouse is not in an idle state (such as reshaping or `movRng` a window). Otherwise, they are handled by the TTY Process.

When these interrupts are taken in the mouse process, they abort the current mouse-invoked window operation. Thus, if you have selected a window operation that requires a "long" computation (such as searching a directory in the `FileBrowser`), you may abort that computation using `<CTRL-E>`.

The `RESET` interrupt causes a process to be unwound to the top level of its stack. If the process is designated as restartable (e.g., its `RESTARTABLE` process property has the value `T`), it is restarted; otherwise, the process is killed.

6.1.4.2 Handling Help Interrupts

The `HELP` interrupt (initially, `<CTRL-G>`) causes a menu of processes to be displayed to you. Using the mouse, you select the

PROCESS MANAGEMENT

process in which you want the interrupt to occur. The current TTY Process is designated by a * next to its name.



Figure 6.1 Depiction of HELP Interrupt Menu

The menu includes an entry "Spawn Mouse" which allows you to create a new mouse process if the current mouse process is tied up executing some processes' BUTTONEVENTFN. When this entry is selected, a new mouse process is created, but no break occurs.

6.1.4.3 Handling Breaks

The BREAK interrupt (initially, <CTRL-B>) causes a Break Window to be opened.

6.1.4.4 Handling Rubout

The RUBOUT interrupt (initially, DELETE) clears the typeahead entries in all processes. That is, its the value of PROCTYPEAHEAD in each processes' object to NIL.

6.1.4.5 Handling Stack and Storage Overflow

PROCESS MANAGEMENT

STACK OVERFLOW and STORAGE FULL interrupts are taken by the process in which the condition occurred since it is likely that this is the process which causes the condition. However, it is not necessarily so, since some other process may have "runaway" and consumed all of the resources before passing control to the process in which the interrupt occurred.

6.1.5 Enabling the Process World

You may enable and disable the process world using **PROCESSWORLD**, which takes the form:

Function:	PROCESSWORLD
# Arguments:	1
Arguments:	1) FLG, a flag.
Value:	T or nothing.

When you initialize Interlisp from the standard sysout, the process world is enabled. You may disable it by:

```
<-(PROCESSWORLD 'OFF)  
<nothing is returned here>
```

which kills all processes and turns process management off. Note that PROCESSWORLD does not return when you disable process management. You may enable process management after it has been turned off via:

```
<-(PROCESSWORLD 'ON)  
T
```

If processes are already enabled, PROCESSWORLD returns the following message:

```
(Processes are already on)
```

Note: PROCESSWORLD is only intended to be called from the top level. When it is invoked, it constructs new processes with new stack space. Previous callers of PROCESSWORLD will be left in an inaccessible part of the stack because there is no reference back to it.

6.1.6 The Process Status Window

You may display a Process Status Window by selecting the menu item PSW from the background menu. Alternatively, you may call the function PROCESS.STATUS.WINDOW to display the Process Status Window.

The Process Status Window consist of two menus as depicted in Figure 6.2. The top menu lists all processes known to Interlisp while the bottom menu provides a list of commands that operate upon a selected process.

TO BE DETERMINED

[fg6.2>The Process Status Window]fg

6.1.6.1 The List of Processes

The top menu of the Process Status Window contains a list of the processes currently known by Interlisp. This list is updated whenever a new process is created or an existing process is destroyed.

6.1.6.2 Process Status Window Commands

The bottom menu of the Process Status Window contains a list of commands which you may select to operate upon processes selected from the top menu. The commands are presented in Table 6-2.

Table 6-2. Process Status Window Commands

Com mand	Action
BT	Displays a backtrace of the selected process.
BTV	Displays a backtrace with variables of the selected process.
BTV*	Displays a backtrace with argument names of the selected process.
BTV!	Displays a backtrace with argument names and values of the selected process.
WHO?	Changes the selection to the TTY process, e.g., the process currently in control of the keyboard.
KBD<-	Associates the keyboard with the selected process, if
INFO	Invokes the function associated with the processes' INFOHOOK property, if one exists; otherwise
BREAK	Causes a break to occur in the current function executed
KILL	Deletes the selected process.
RESTART	Restarts the selected process, if it is restartable.
WAKE	Wakes the selected process after prompting you to supply a value with which to wake the process.
SUSPEND	Suspends the selected process, i.e., causes the process to be blocked indefinitely until it is explicitly awoken.

6.1.6.3 Opening the Process Status Window

If the Process Status Window is not open, you may open it explicitly from within your program by calling `PROCESS.STATUS.WINDOW`, which takes the form:

Function:	<code>PROCESS.STATUS.WINDOW</code>
# Arguments:	1
Arguments:	1) WHERE, a position
Value:	The window handle for the Process Status Window.

`PROCESS.STATUS.WINDOW` displays a Process Status Window if one is not already open. If a Process Status Window is open, it is merely refreshed.

WHERE is a position on the screen where the Process Status Window will be displayed. Otherwise, you will be prompted for a location at which to place the Process Status Window.

6.2 Creating and Destroying Processes

You may dynamically create and destroy processes in Interlisp. Care should be exercised in destroying processes because all objects used by the process may not be eliminated. Thus, you may start to lose portions of virtual memory due to objects which have not been garbage collected.

6.2.1 Creating a Process

You may create a new process that evaluates an expression using `ADD.PROCESS`, which takes the form:

PROCESS MANAGEMENT

Function:	ADD.PROCESS
# Arguments:	1-N
Arguments:	1) FORM, an expression to be evaluated 2) PROP, a process property 3) VALUE, a process property value 4-N) PROP/VALUE pairs
Value:	A process handle.

ADD.PROCESS is a nospread function. It creates a new process which evaluates FORM and returns the process handle.

The new process is created with a new stack environment. Thus, it does not have access to the stack environment of the process which created it. All information to be passed to the new process must be passed as arguments in FORM. Alternatively, FORM invokes functions which know about global variables that permit sharing between the new process and its parent.

The new process runs until FORM returns either via executing PROCESS.RETURN or exiting a function, or the process is explicitly deleted by another process. A process may be terminated if an error occurs which is not trapped by the user.

The PROP/VALUE pairs allow you to set properties of the process. Each PROP/VALUE pair is given to PROCESSPROP to store on the property list of the process handle.

Two process properties are particularly relevant to ADD.PROCESS:

1. NAME: Its value should be a literal atom which will be the name of the process. If NIL, the name is taken from FORM.
2. SUSPEND: If its value is non-NIL, the new process is created but immediately suspended. The process will not be run until it is awakened by a call to WAKE.PROCESS.

PROCESS MANAGEMENT

ADD.PROCESS will attempt to make the name of the process unique by packing it with a number. The name of a process is used to manipulate the process when you type in expressions at the top-level. That is, the name is given to various process world functions or it appears in menus of processes.

Caution should be exercised because the relationship between process names and process handles is sometimes tenuous. Process handles are unique and are handled properly by Interlisp.

Consider the following examples:

```
<-(SETQ XP (ADD.PROCESS '(PROCESS.DEMO)))  
{PROCESS}#77,110200
```

where we define PROCESS DEMO as:

```
(DEFINEQ (PROCESS.DEMO NIL  
  (PROG (PW)  
    (SPAWN.MOUSE)  
    (SETQ PW (GENSYM 'PW))  
    (SET PW (CREATEW NIL "PROCESS  
DEMO WINDOW"))  
    (for X from 1 by 1 to 10000  
      do  
        (SETQ A (IPLUS A 1))  
        (PRIN1 A PW)  
        (TERPRI PW)  
        (DISMISS 1000)  
        (if (GREATERP A 10)  
          then  
            (PROGN  
              (CLEARW PW)  
              (SETQ A 0)
```


PROCESS MANAGEMENT

```
(PRIN1 "STARTING COUNT
ANEW"
PW)
    (TERPRI PW)
    (AWAIT.EVENT EV1))))
(PRIN1 "ALL DONE NOW" PW)
(TERPRI PW)
(PROCESS.RETURN (APPEND (LIST "MY
PROCESS ID +"
                        (THIS.PROCESS))
                    )))
(PROCESS.DEMO)
```

This is a very simple process declaration. The idea is to demonstrate the use of processes in the following examples. When the process is created, the window appears as depicted in Figure 6.2 (after running for a short time).

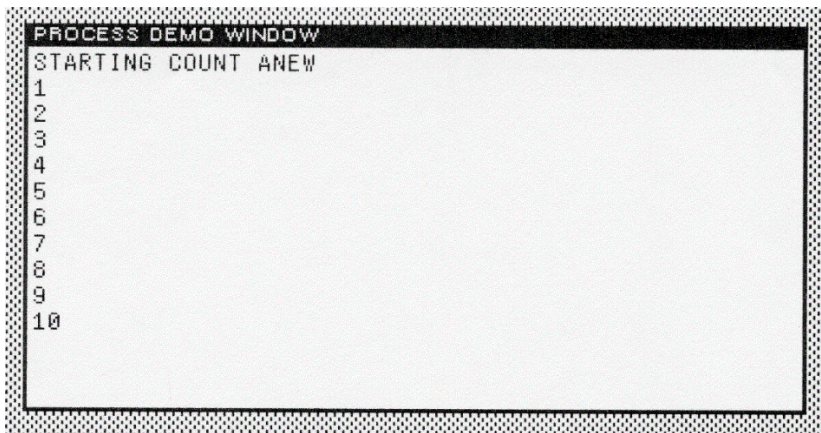


Figure 6.2 PROCESS DEMO WINDOW

PROCESS MANAGEMENT

6.2.2 Killing a Process

You may kill a process by executing **DEL.PROCESS**, which takes the form:

Function:	DEL.PROCESS
# Arguments:	1
Arguments:	1) PROC, a process handle
Value:	T

You may delete (i.e., kill) a process which is currently running or suspended via DEL.PROCESS. PROC may be a process handle or its name. Consider the example:

```
<-(SETQ XP (ADD.PROCESS '(PROCESS.DEMO)))  
{PROCESS}#56,56100  
  
<-(DEL.PROCESS XP)  
T
```

After the process is killed, the Process Status Window shades the name of the process to indicate it has been killed (see Figure 6.3).

TBD

[fg6.3>Process Status Window after DEL.PROCESS]fg

If PROC is the currently running process, DEL.PROCESS does not return (for where would it turn to?). Executing DEL.PROCESS at the top level can be potentially fatal to your Interlisp environment.

6.3 Process Properties

PROCESS MANAGEMENT

A process may be described by a number of process properties. We have already mentioned two of these - NAME and SUSPEND - in the discussion of ADD.PROCESS above.

6.3.1 Getting and Setting Process Properties

You may get or set process properties using the function **PROCESSPROP**, which takes the form:

Function:	PROCESSPROP
# Arguments	3
Arguments:	1) PROC, a process handle or name 2) PROP, a property name 3) NEWVALUE, a value for the property
Value:	The value of the processes' property.

PROCESSPROP is a nospread function. PROCESSPROP gets or sets the values of properties that are stored on the process handle's property list. It returns the following value according to the functions performed:

- The current value of the property if NEWVALUE is not supplied, or
- The old value of the property if NEWVALUE is supplied, including if it is NIL.

Thus, what distinguishes retrieval from storage is whether or not NEWVALUE is supplied in the argument list. Consider the following examples:

```
<-MYPROCESS  
{PROCESS}#71,22400
```

```
<-(PROCESSPROP MYPROCESS 'NAME)  
TEdit
```

```
<-(PROCESSPROP MYPROCESS 'WINDOW)
{WINDOW}#55,141404
```

Certain properties have special meanings for processes. These are described in Sections 6.3.2 through 6.3.9. All other properties are ignored by the process management functions.

6.3.2 Process Name

As mentioned in Section 6.2.1, you may assign a name to a process which allows you to identify it to process management functions. The name, which must be a literal atom, is stored under the property NAME on the process handle. If the name is not unique, ADD.PROCESS makes the name unique by packing it with a number.

In general, I recommend that you select names which are meaningful because the names of processes appear in different menus throughout the system.

6.3.3 The Process Body

The original expression used to initialize the process is stored on the process handle under the property FORM. Consider the following example:

```
<-(SETQ XP (ADD.PROCESS '(PROCESS.DEMO)))
{PROCESS}#56,56100
```

```
<-(PROCESSPROP XP 'FORM)
(PROCESS.DEMO)
```

Note that the form for the Exec Window is (\PROC.REPEATEDLYEVALQT) which you can determine by inspecting the stack after breaking in the Exec Window.

6.3.4 The Restart Flag

A process may be deleted if an error occurs which causes its body to be exited during execution. The property `RESTARTABLE` determines the disposition of the process object. It may take the values presented in Table 6-3.

Table 6-3. RESTARTABLE Values

Value	Meaning
NIL or NO	If an untrapped error, a CTRL-E, or a CTRL-D occurs, the process body is exited and the process is deleted.
T or YES	The process will be automatically restarted when an error occurs or CTRL-D is pressed. Most system processes, such as the mouse process, have this as the default value.
HARDRESET	The process will be deleted if an error occurs which causes its body to be exited, but is restarted on a hard reset (such as CTRL-D).

The preferred setting for persistent processes is `HARDRESET` when errors are largely unexpected, because the error may simply re-occur when the process is restarted.

Needless to say, caution should be exercised in determining when processes should be restarted automatically, particularly those which write to external files.

6.3.5 A Restart Form

When a process is restarted, it will typically re-execute the original form which is stored as the value of `FORM`. However, automatic restart may occur in an uncertain state where the conditions expected

PROCESS MANAGEMENT

by the original form are no longer valid. You may specify an alternative form to be executed upon restart as the value of the property **RESTARTFORM**.

6.3.6 Processing Before an Exit

You may have processes running when you call **LOGOUT**. Such an event could leave you sysout in an uncertain state, especially if certain operations are time-dependent. You may specify processing to be performed prior to exiting as the value of the property **BEFOREEXIT**.

If its value is **DON'T**, the process will not be interrupted by the **LOGOUT** procedure. If **LOGOUT** is attempted before the process finishes, a message will appear which informs you that Medley Interlisp is waiting for the process to finish before proceeding with the logout.

6.3.7 Processing After an Exit

After a system exit, you may specify processing for a process by assigning a value to the property **AFTEREXIT**. Values which it may take are presented in Table 6-4.

Table 6-4. AFTEREXIT Values

Value	Meaning
DELETE	The process is deleted.
SUSPEND	The process is suspended and not run until explicitly awoken.
An event	The process is suspended waiting for an event to occur.

6.3.8 An Information Hook

PROCESS MANAGEMENT

You may want to query a process about its status using the `INFO` command that appears in the Process Status Window. The value of `INFOHOOK` is a function which executes within the stack environment of the process. Thus, it can provide additional information about the process.

6.3.9 Handling the TTY Display Stream

As mentioned in Section 6.1 above, a process may assume the role of the TTY display stream process. Two properties, `TTYENTRYFN` and `TTYEXITFN`, can have as their values functions which are invoked whenever the current process assumes the role of the TTY display stream or ceases to have that role.

6.4 Process Management Functions

Medley Interlisp provides a number of functions for managing the process environment. When coupled with the event management functions and the monitor locking functions, Interlisp provides a complete, albeit primitive, process control environment. But, this environment gives you the ability to build sophisticated process control environments that suit your applications.

6.4.1 Testing for an Active Process

You may test if a given process is an active process (e.g., neither waiting nor suspended) using the function **PROCESSP**, which takes the form:

Function	PROCESSP
# Arguments:	1
Arguments:	1) PROC, a process handle
Value:	T.

PROCESS MANAGEMENT

PROCESSP determines whether the given process has finished executing or not. Consider the following example:

```
<-MYPROCESS  
{PROCESS}#77,110100  
  
<-(PROCESSP MYPROCESS]  
T
```

which verifies that MYPROCESS is a process handle.

6.4.2 Testing for a Deleted Process

You may test whether a process handle is that of a deleted process using the function **RELPROCESSP**, which takes the form:

Function:	RELPROCESSP
# Arguments:	1
Arguments:	1) PROHANDLE, a process handle
Value:	T, if the handle is of a deleted process.

Consider the following examples:

```
<-(SETQ XP (ADD.PROCESS '(PROCESS.DEMO)))  
{PROCESS}$56,56600  
  
<-(DEL.PROCESS XP)  
T  
  
<-(PROCESS.FINISHEDP XP)  
DELETED  
  
<-(REPROCESSP XP)  
T
```


6.4.3 Testing for a Finished Process

You may test whether or not a process has completed execution using the function **PROCESS.FINISHEDP**, which takes the form:

Function:	PROCESS.FINISHEDP
# Arguments	1
Argument:	1) PROCESS, a process handle
Value:	An indication of the termination condition if the process has completed; otherwise, NIL.

PROCESS.FINISHEDP returns either of the atoms NORMAL or ERROR to indicate how a process terminated, if, indeed, it has terminated. It returns the atom DELETED if the process has been killed either from the menu in the Process Status Window or through invocation of DEL.PROCESS. Consider the following example:

```
<-(SETQ XP (ADD.PROCESS '(PROCESS.DEMO)))
{PROCESS}$56,56600
```

```
<-(DEL.PROCESS XP)
T
```

```
<-(PROCESS.FINISHEDP XP)
DELETED
```

Now, here is an example where the process completes normally:

```
<-(SETQ XP (ADD.PROCESS '(for X from 1 to 1000 do (SETQ A X))))
{PROCESS}#77,110500
```

```
<-(PROCESS.FINISHEDP XP)
NORMAL
```

6.4.4 Finding a Process Handle

Within a process you may need to determine the process handle of another process for any number of reasons: to send it a message, to delete it or restart it, or just to retrieve information about it. Several functions have been provided to assist you in finding a processes' handle.

6.4.4.1 Finding the Current Processes' Handle

You may find the handle of the currently running process using the function `THIS.PROCESS`, which takes the form:

Function:	<code>THIS.PROCESS</code>
# Arguments:	0
Arguments:	N/A
Value:	The process handle of the current process.

`THIS.PROCESS` returns the handle of the current process. Consider the following example:

```
<-(THIS.PROCESS)
{PROCESS}#71,22400
```

which happens to be the process handle of the Exec process.

Note that the currently running process may have no information concerning its handle unless that information was passed to it after it was initialized and execution began.

If the Process World is disabled, `THIS.PROCESS` returns `NIL` (e.g., it acts like a no-operation).

6.4.4.2 Finding Any Processes' Handle

You may determine the process handle of a process using the function `FIND.PROCESS`, which takes the form:

Function:	<code>FIND.PROCESS</code>
# Arguments:	2
Arguments:	1) <code>PROC</code> , a process handle or name 2) <code>ERRORFLG</code> , an error flag
Value:	A process handle.

`FIND.PROCESS` returns a process handle if `PROC` is a process handle or is a process name. Of course, the latter case is the most useful since many processes will initially know of other processes by their names. Consider the following examples:

```
<-(FIND.PROCESS 'MOUSE)
{PROCESS}#56,56000
```

```
<-(FIND.PROCESS 'BACKGROUND)
{PROCESS}#71,22600
```

`ERRORFLG` determines how `FIND.PROCESS` responds if `PROC` does not name an active process. If `T`, it causes an error to be generated.

6.4.5 Restarting a Process

You may force a process to restart using the function `RESTART.PROCESS`, which takes the form:

Function:	<code>RESTART.PROCESS</code>
# Arguments:	1
Arguments:	1) <code>PROC</code> , a process handle
Value:	The process handle.

RESTART.PROCESS unwinds the given process to its top level and reevaluate its form. The effect is to delete the process (as if DEL.PROCESS were called) and then add a new process (as if ADD.PROCESS were called).

6.4.6 Returning a Value from a Process

Interlisp allows you to retrieve the result from running a process in two ways: either as a function call or as an independent coroutine.

6.4.6.1 Returning a Value

A process may return a value which is the result of its execution. It does so by invoking the function PROCESS.RETURN, which takes the form:

Function:	PROCESS.RETURN
# Arguments:	1
Argument:	1) VALUE, the value to be returned
Value:	The value given above.

When a process is created by ADD.PROCESS, an implicit call to PROCESS.RETURN is wrapped around the form to be evaluated. Thus, a process normally finishes by simply returning, e.g., by executing the last statement of its form.

PROCESS.RETURN terminates the current process and explicitly returns the value specified as the argument. It is provided for causing earlier termination of a process such as the occurrence of an error in the computation.

6.4.6.2 Obtaining a Processes' Result

PROCESS MANAGEMENT

A process may return a value resulting from its execution as if it were called like a function. You may obtain the result returned by a given process using the function `PROCESS.RESULT`, which takes the form:

Function:	<code>PROCESS.RESULT</code>
# Arguments:	2
Arguments:	1) <code>PROC</code> , a process handle 2) <code>WAITFORRESULT</code> , a flag
Value:	The result from the process; otherwise, <code>NIL</code> .

`PROCESS.RESULT` returns the value returned from `PROC` if it has terminated. This value is returned either via `PROCESS.RETURN` or from the `ADD.PROCESS` expression.

If the process aborted its execution, i.e., returned an error, the value returned is `NIL`.

Note that `PROC` must be a process handle, not a process name. This is because the association between a process name and a process handle is dissolved when a process terminates execution. The process handle may continue to exist if there are pointers to it, but it will eventually be garbage-collected.

`WAITFORRESULT` determines whether or not the caller waits for the result. When `PROCESS.RESULT` is invoked, the target process may not have completed its computation. If `WAITFORRESULT` is `T`, `PROCESS.RESULT` blocks until `PROC` finishes executing. Otherwise, it returns `NIL` immediately.

6.4.7 Mapping Across Processes

You may apply a function to all processes using the function `MAP.PROCESSES`, which takes the form:

PROCESS MANAGEMENT

Function:	MAP.PROCESSES
# Arguments:	1
Arguments:	1) MAPFN, a mapping function
Value:	NIL

MAP.PROCESSES applies MAPFN to all known processes. MAPFN is a function of three arguments:

1. the process handle;
2. the process name; and
3. the process form.

Consider the following example where I have defined PRINT.PROCESS as a rather simple function:

```
<-(MAP.PROCESSES (FUNCTION PRINT.PROCESS))
The process name is: EXEC
The process name is: MOUSE
The process name is: \DLRS232C.WATCHER
The process name is: \10MBWATCHER
The process name is: \NSGATELISTENER
The process name is: \TIMER.PROCESS
The process name is: BACKGROUND
NIL
```

where PRINT.PROCESS is defined as:

```
(DEFINEQ (PRINT.PROCESS (X)
  (PRIN1 "The process name is:")
  (SPACES 2)
  (PRIN1 (PROCESSPROP X 'NAME))
  (TERPRI)
))
(PRINT.PROCESS)
```

6.5 Process Control Functions

Interlisp provides a standard set of process control functions that allow you to control the execution of individual processes according to the availability of data, the state of the computation, or the time of day.

6.5.1 Suspending a Process

You may suspend a process using the function `SUSPEND.PROCESS`, which takes the form:

Function:	<code>SUSPEND.PROCESS</code>
# Arguments:	1
Arguments:	1) <code>PROC</code> , a process handle
Value:	The process handle.

`SUSPEND.PROCESS` blocks the specified process indefinitely, i.e., it will not be scheduled for execution until it is awoken by another process. Consider the following example:

```
<-(SUSPEND.PROCESS XP)
{PROCESS}#55,56400
```

You may cause the current process to block indefinitely using the following form:

```
<-(SUSPEND.PROCESS (THIS.PROCESS))
```

6.5.2 Awakening a Process

When a process is blocked, it is not executable. You may block a process (as explained below) for a short time or indefinitely. A process blocked on a timer is awoken when its timer expires. Suspended

PROCESS MANAGEMENT

process do not awake unless explicitly awoken by another process. You may awake a process which is blocked using the function WAKE.PROCESS, which takes the form:

Function:	WAKE.PROCESS
# Arguments:	2
Arguments:	1) PROC, a process handle 2) STATUS, a status to be returned
Value:	T

WAKE.PROCESS explicitly awakens the specified process and causes it to call BLOCK with no arguments to obtain its status. When the process that was awakened calls BLOCK, it will receive the value of STATUS as its status. The process is scheduled for execution, but will not executed until it arrives at the top of the schedule queue.

WAKE.PROCESS will awaken a process blocked on a timer before the timer expires. Consider the following example:

```
<-(SUSPEND.PROCESS XP)  
{PROCESS}#55,107600
```

```
<-(PROCESSPROP XP 'STATUS)  
0
```

```
<-(WAKE.PROCESS XP 100)  
T
```

You may invoke WAKE.PROCESS on a blocked process many times before it is actually executed. In this case, it receives only the most recent STATUS.

6.5.3 Blocking a Process

PROCESS MANAGEMENT

A process may be blocked on a timer for a specified period of time using the function `BLOCK`, which takes the form:

Function:	<code>BLOCK</code>
# Arguments:	2
Arguments:	1) <code>MSECWAIT</code> , milliseconds to wait 2) <code>TIMER</code> , an interval timer
Value:	A status.

When `BLOCK` is called, the current process yields control to the next process which is ready to execute, assuming any is ready to run. If `BLOCK` is called without any arguments, the process remains executable, but is forced to the end of the scheduling queue.

If `MSECWAIT` is specified, it is the number of milliseconds to wait before rescheduling the process. Consider the following example:

If `MSECWAIT` is `T`, the process waits indefinitely, e.g., it is suspended. In this case, the process must be explicitly awoken via `WAKE.PROCESS`.

`TIMER` can be specified as a millisecond timer which is created by `SETUPTIMER`. In this case, `TIMER` is an absolute time at which the process will be awoken.

The effect of `BLOCK` may be terminated by `CTRL-D`, `CTRL-E`, or `CTRL-B` as well.

```
<-(BLOCK 1000)
"{timer interval expired}"
```

when executed in the Exec Window.

6.5.4 Dismissing a Process

PROCESS MANAGEMENT

You may dismiss the current process for a given period of time, e.g., force it to be rescheduled some relative time in the future using the function `DISMISS`, which takes the form:

Function:	<code>DISMISS</code>
# Arguments:	3
Arguments:	1) <code>MSECSWAIT</code> , the milliseconds to wait 2) <code>TIMER</code> , an interval timer 3) <code>NOBLOCK</code> , a flag
Value:	The value of <code>MSECWAIT</code> .

When you run the example of `PROCESS.DEMO` given above in multiple windows, you will see the windows stop and start printing at different times. This is because after every `SETQ` of `A` (see the code for `PROCESS.DEMO`), we do an immediate `DISMISS`. More interesting effects can be accomplished by dismissing for a random amount of time.

6.5.5 Evaluating Expressions in a Processes' Context

Each process has its own context stack. Usually, expressions are evaluated within the context of a specific process. There are often general expressions applicable across a set of processes which can be executed in any processes' context. Interlisp provides three functions for evaluating arbitrary functions in a processes' context.

6.5.5.1 Evaluating a Variable in Context

You may evaluate a variable within a processes' context using the function `PROCESS.EVALV`, which takes the form:

Function:	<code>PROCESS.EVALV</code>
# Arguments:	2
Arguments:	1) <code>PROC</code> , a process handle

PROCESS MANAGEMENT

Value: 2) VAR, a variable name
 The value of VAR in the processes'
 context.

PROCESS.EVALV evaluates the variable within the processes' context by executing (EVALV VAR) in the stack context of PROC. Consider the following example in the environment of the process represented by XP (actually an instance of PROCESS.DEMO):

```
<-XP
{PROCESS}#56,56100

<-(PROCESS.EVALV XP 'X)
300
```

after XP has been executing for awhile.

6.5.5.2 Evaluating an Expression in a Processes' Context

You may evaluate an expression within a processes' context using the function **PROCESS.EVAL**, which takes the form:

Function: PROCESS.EVAL
Arguments: 3
Arguments: 1) PROC, a process handle
 2) FORM, an expression to be evaluated
 3) WAITFORRESULT, a flag
Value: The value of the expression.

PROCESS.EVAL evaluates FORM in the stack context of PROC. All variables occurring in FORM are bound within the context of PROC. Consider the following example:

```
<-(PROCESS.EVAL XP 'X)
320
```

```
<-(PROCESS.EVAL XP '(SETQ X 400))
400
```

```
<-(PROCESS.EVAL XP 'X)
402
```

Any errors that occur in evaluating FORM will occur within the context of PROC. Thus, they must be handled within that context. The IRM notes that the following forms will achieve different results:

```
(PROCESS.EVAL PROC '(NLSETQ <form>))
```

```
(NLSETQ (PROCESS.EVAL PROC <form>))
```

If WAITFORRESULT is true, the current process is blocked until the evaluation is completed. Otherwise, the current process continues to run in parallel with the evaluation.

6.5.5.3 Applying a Function to Arguments in a Context

You may apply a function to a set of arguments within a processes' context using the function PROCESS.APPLY, which takes the form:

Function:	PROCESS.APPLY
# Arguments:	4
Arguments:	1) PROC, a process handle 2) FN, a function 3) ARGS, a list of arguments 4) WAITFORRESULT, a flag
Value:	The value of FN when applied to the arguments.

PROCESS.APPLY applies the function FN to the argument list within the stack context of PROC. If the process is currently

suspended, Interlisp will print the message "<process handle> not a live process".

6.6 Interprocess Communication

An *event* is a synchronizing mechanism used to coordinate the actions of two or more processes. The typical paradigm used is the "producers and consumers" model familiar to most users through their study of operating systems. In this model, consumers *wait* on events while producers *notify* events.

Interlisp provides a simple event mechanism that allows you to coordinate the actions of processes within your program. Every event has the structure presented in Table 6-5.

Table 6-5. Event Structure

Field	Usage
EVENTNAME	The name given to CREATE.EVENT.
EVENTQUEUETAIL	Pointer to the next event.
EVENTWAKEUPPENDING	A list of processes waiting on the event.

You are free to choose the meaning of each event that you create. Generally, the idea of an event is to let some process or set of processes know that something interesting has happened which they should examine. It is up to the receiving processes to determine the significance of the event.

Processes may be awoken from waiting on an event either by the occurrence of the event or because the waiting period expired (if specified). You must write your processes so that they can handle either case correctly.

PROCESS MANAGEMENT

Note that Interlisp does not provide for the passing of messages through events unlike other event mechanisms. The Interlisp event mechanism is a pure signalling mechanism.

6.6.1 Creating an Event

You may create an event using the function `CREATE.EVENT`, which takes the form:

Function:	<code>CREATE.EVENT</code>
# Arguments:	1
Arguments:	1) <code>NAME</code> , the name of the event
Value:	An event object handle.

`CREATE.EVENT` builds a data structure that represents the event and links it into the event queue. `NAME` is an arbitrary value chosen by you which may be used for obtaining status information or for debugging. Consider the following example:

```
<-(SETQ EV1 (CREATE.EVENT 'EVENT-1))
{EVENT}#54,110650
```

6.6.2 Awaiting an Event

A consumer process awaits the occurrence of an event before proceeding with its computation in the typical "producers and consumers" model. You may cause a process to wait for an event by executing `AWAIT.EVENT`, which takes the form:

Function:	<code>AWAIT.EVENT</code>
# Arguments:	3
Arguments:	1) <code>EVENT</code> , the handle of an event 2) <code>TIMEOUT</code> , the waiting period 3) <code>TIMERP</code> , a millisecond timer
Value:	The value of <code>EVENT</code> .

PROCESS MANAGEMENT

`AWAIT.EVENT` suspends the current process until the given event is notified or until a timeout occurs. It returns the handle of the event.

If `TIMEOUT` is `NIL`, there is not timeout period and the process waits until `EVENT` is notified, but possibly forever. `TIMEOUT` should be a number representing the milliseconds to wait.

If `TIMERP` is non-`NIL`, it is a millisecond timer set to expire at the time specified when it is created by `SETUPTIMER`.

Consider the following example:

```
<-(SETQ XP (ADD.PROCESS '(PROCESS.DEMO)))  
{PROCESS}#54,106500
```

Now, the process has blocked on `EV1` because it has executed the statement (`AWAIT.EVENT EV1`). Because `TIMEOUT` is `NIL`, the process will wait forever or until a `NOTIFY.EVENT` call is issued to restart it.

6.6.3 Signalling Completion of an Event

A producer "notifies" an event when it wants to signal one or more other processes that they may proceed with corollary computations. A process notifies an event using the function `NOTIFY.EVENT`, which takes the form:

Function:	<code>NOTIFY.EVENT</code>
# Arguments:	2
Arguments:	1) <code>EVENT</code> , an event handle 2) <code>ONCEONLY</code> , a flag
Value:	<code>NIL</code>

NOTIFY.EVENT causes those processes waiting for event to be awakened and placed in an executable state. Each process is awoken with the value of EVENT.

If ONCEONLY is true, only the first process waiting on the event will be awoken and placed in the executable state.

Care should be exercised in using this option. It is your responsibility to ensure that only one process can respond to the event at once or, if multiple processes can respond, that the order of their responding does not affect the computation.

Continuing with the example above, I notify event EV1 upon which process XP is waiting as follows:

```
<-(NOTIFY.EVENT EV1)
NIL
```

which causes XP to be awoken.

6.7 Monitors: Sharing Data Structures

Cooperating processes often need to share information during a computation. The event mechanism permits processes to signal each other when some action should be performed, but does not provide a mechanism for communicating information. Hoare defined the concept of *monitors* to provide shared, but mutually exclusive, access to data structures by more than one process.

Interlisp has implemented a simple monitor mechanism which allows two or more processes to share a data structure on a mutually exclusive basis. Monitors are represented by objects called *monitor locks*, which are created by a process and associated with some data structure which is shared, but must be protected from simultaneous access.

Monitor locks have the data structure presented in Table 6-6.

Table 6-6. Monitor Lock Structure

Field	Usage
MLOCKLINK	Link to next monitor lock in the list.
MLOCKNAME	The name of the monitor lock.
MLOCKOWNER	The handle of the process creating the monitor lock.
MLOCKQUEUETAIL	The last monitor lock in the queue.
MLOCKPERPROCESS	The number of monitor locks per process.

When a process is deleted, any locks it owns are released.

6.7.1 Creating a Monitor

You may create a monitor lock using the function **CREATE.MONITORLOCK**, which takes the form:

Function:	CREATE.MONITORLOCK
# Arguments:	1
Arguments:	1) NAME, the name of the monitor lock
Value:	A monitor lock handle.

CREATE.MONITORLOCK builds a data structure which contains the information about the monitor and returns the handle of the monitor lock object. **NAME** is used for obtaining status information or for debugging.

Consider the following example:

```
<-(SETQ ML1 (CREATE.MONITORLOCK 'LOCK-
1))
{MONITORLOCK}#71,16570
```

6.7.3 Evaluating Expressions under a Monitor Lock

Medley Interlisp provides two mechanisms for using monitor locks which may be termed the slow and fast versions. The primary difference is that the slow version is implemented using RESETLST in order to handle errors that may occur, while the fast version is not. These two mechanisms take the form:

Macro:	WITH.MONITOR WITH.FAST.MONITOR
# Arguments:	2-N
Arguments:	1) LOCK, a monitor lock handle 2-N) FORM1 - FORMn, expressions
Value:	The value from evaluating the last form.

Each of these macros evaluates the specified expressions while owning the given monitor lock. Ownership of the lock is dynamically scoped, e.g., if the current process already owns the lock, evaluation of the expressions proceeds forthwith. This may occur when a function is called by another function within the process which was already executing within a WITH.MONITOR macro for the specified monitor lock.

The RESETLST is used to free the lock if an error occurs while any of the FORMs are being evaluated.

WITH.MONITOR takes the form:

```
((LOCK . FORMS)
(RESETLST
```

PROCESS MANAGEMENT

```
(OBTAIN.MONITORLOCK LOCK NIL T)
(PROGN . FORMS)
))
```

WITH.FAST.MONITOR acts like WITH.MONITOR, but is not implemented with a RESETLST. User interrupts (e.g., CTRL-E) are inhibited while executing within this macro. Moreover, since there is no error protection, the specified forms must never terminate in error because the lock will not be released. Thus, this macro is often used for small, safe computations which are error-free and are non-interruptable. WITH.FAST.MONITOR takes the form:

```
((LOCK . FORMS)
 (UNINTERRUPTABLY
  ((LAMBDA (UNLOCK)
   (PROG1
    (PROGN . FORMS)
    (AND
     (NEQ UNLOCK T)
     (RELEASE.MONITORLOCK UNLOCK)))
   ))
  (OBTAIN.MONITORLOCK LOCK)
 ))
))
```

6.7.4 Awaiting a Monitor Event

When one process is accessing a shared data structure, all other processes should wait until the data structure is free. Thus, cooperating processes must signal each other when they have completed operation upon a shared data structure. You may use the function **MONITOR.AWAIT.EVENT**, which takes the form:

Function:	MONITOR.AWAIT.EVENT
# Arguments:	4

PROCESS MANAGEMENT

Arguments: 1) RELEASELOCK, a monitor lock handle
 2) EVENT, an event handle
 3) TIMEOUT, a waiting period (optional)
Value: NIL

MONITOR.AWAIT.EVENT is used to block processing inside a monitor. It releases the lock and then executes a call to AWAIT.EVENT. When the process is awoken, it reobtains the lock.

MONITOR.AWAIT.EVENT is often used when a process wants to perform an operation on a data structure, but expects the data structure to have a certain configuration or state. It uses a monitor lock to ensure that the state of the structure does not change between the time it tests the state and when it performs the operation. If the state is not the correct one, the process can wait for some other process to make it the correct state, whence it can proceed with the operation. In the mean time, it releases the lock so that other processes can have access to the data structure (including the process which will make the state correct).

The form for this usage might appear as (according to the IRM):

```
(WITH.MONITOR <object>-LOCK
  (until <test condition of object>
    do
      (MONITOR.AWAIT.EVENT <object>-LOCK
        <object>-CHANGED-EVENT
        <object>-TIMEOUT)
        <operate on object>)
  )
)
```

PROCESS MANAGEMENT

The IRM suggests that this form is "cleaner" (read "more efficient") because it saves the RESETLST processing performed in WITH.MONITOR.

The IRM notes that there must not be an ERRORSET invocation between the enclosing WITH.MONITOR and the call to MONITOR.AWAIT.EVENT, because such an ERRORSET would catch any calls to ERROR! and continue within the monitor. This is not the condition you want since the monitor lock is not reobtained.

6.7.5 Obtaining a Monitor Lock

A process may take possession of a monitor lock, waiting if necessary until it is free, using the function **OBTAIN.MONITORLOCK**, which takes the form:

Function:
OBTAIN.MONITORLOCK
Arguments: 3
Arguments: 1) LOCK, a monitor lock
 handle
 2) DONTWAIT, a flag
 3) UNWINDSAVE, a flag
Value: The monitor lock handle.

OBTAIN.MONITORLOCK takes possession of LOCK, but waits until it is free. It returns the monitor lock handle if the lock is successfully obtained. If the process already owns the monitor lock, it returns T.

If DONTWAIT is non-NIL, OBTAIN.MONITORLOCK returns NIL immediately.

If UNWINDSAVE is non-NIL, it wraps the action of taking the monitor lock in a RESETSAVE expression.

6.7.6 Releasing a Monitor Lock

A process may release a monitor lock using the function **RELEASE.MONITORLOCK**, which takes the form:

Function:
RELEASE.MONITORLOCK
Arguments: 2
Arguments: 1) LOCK, a monitor lock
 handle
 2) EVENIFNOTMINE, a
 flag
Value: NIL

RELEASE.MONITORLOCK releases the specified lock if it is owned by the current process. When a lock is released the next waiting process, which is waiting to obtain the lock, is awoken.

EVENIFNOTMINE, if non-NIL, is a flag which permits a process to release a lock even if it does not own it. This flag is useful for freeing "frozen" locks where the evaluation within the lock has somehow failed without freeing the lock. It should be used judiciously.

Index

- ADD.PROCESS**, 310
AREA.OF.RECTANGLE,
 280
 AWAIT.EVENT, 334
 Bravo Text Editor, 14
 Break Package, 191
 AUTOBACKTRACEFLG,
 159
 BREAKREGIONSPEC,
 160
 MaxBkMenuHeight, 158
 BreakPackage
 CLOSEBREAKWINDOW
 FLG, 160
 MaxBkMenuWidth, 158
BRUSHBITMAP, 255
CENTER, 278
 CREATE.EVENT, 334
CREATE.MONITORLOC
 K, 337
curve, 265
 DC, 93
 DEdit, 88
DEDITCOMS, 108
DEDITIT, 95
DEFAULT.INSPECTW.PR
 OPCOMMANDFN, 181
DEFAULT.INSPECTW.TI
 TLECOMMANDFN, 182
DEL.PROCESS, 314
 DF, 89
 DISMISS, 330
DISPLAY.RECTANGLE,
 274
 DISPLAYHELP, 117
DP, 92
DRAWBETWEEN, 264
DRAWCIRCLE, 281
DRAWCURVE, 265
DRAWELLIPSE, 283
DRAWGRAYBOX, 268
DRAWLINE, 262
DRAWPOINT, 248
DRAWTO, 258
 DUMPGRAPH, 240
 DV, 90
 EDITGRAPH, 230
 EDITGRAPH1, 231
EXPAND.RECTANGLE,
 279
 Face Menu, 29
FILLCIRCLE, 288
FILLPOLYGON, 285
 FIND.PROCESS, 323
 FLIPNODE, 237
 Font Menu, 28
GIVE.TTY.PROCESS, 301
graph, 197

INDEX

Graph

DEFAULT.GRAPH.NOD
ELABELSHADE, 206
GRAPH.ADDLINKFN,
201
GRAPH.ADDNODEFN,
200
GRAPH.DELETELINKF
N, 202
GRAPH.MOVENODEFN,
199
TOPJUSTIFYFLG, 213

Grapher

DIRECTEDFLG, 198
GRAPH.DELETENODEF
N, 201
GRAPHEROBJ, 235
NODECREATE, 207
SHOWGRAPH, 211

GRAPHREGION, 236

HASTTYWINDOWP, 302

HEIGHT, 277

INSPECT, 162

inspect window, 170

Inspect Window

FETCHFN, 178
PROPCOMMANDFN,
180
PROPERTIES, 175
PROPPRINTFN, 183
SELECTIONFN, 183
STOREFN, 179
TITLE, 175
VALUECOMMANDFN,
181

Inspect Window

TITLECOMMANDFN,
182

INSPECT/ARRAY, 168,
193

INSPECTCODE, 164

Inspector, 161

INSPECTMACROS, 194

INSPECTPRINTLEVEL,
193

MAXINSPECTARRAYL
EVEL, 193

MAXINSPECTCDRLEVE
L, 193

INSPECTW, 172

INSPECTW.CREATE, 172

INSPECTW.FETCH, 190

INSPECTW.PROPERTIES
, 188

INSPECTW.REDISPLAY,
184

INSPECTW.REPLACE,
184

**INSPECTW.SELECTITE
M**, 186

INSTALLBRUSH, 248

LAYOUTFOREST, 232

LAYOUTGRAPH, 218, 233

LAYOUTSEXPR, 233

LOGOUT, 318

LOWERLEFT, 273

MAP.PROCESSES, 325

monitor lock, 340

monitor locks, 336

INDEX

MONITOR.AWAIT.EVEN
T, 339
monitors, 336
NOTIFY.EVENT, 335
OBTAIN.MONITORLOC
K, 341
OPENTEXTSTREAM, 65
PAINTW.READBRUSHSH
ADE, 252
PAINTW.READBRUSHSH
APE, 250
PAINTW.READBRUSHSI
ZE, 251
process, 290
process handle, 290
Process Macro
WITH.FAST.MONITOR,
339
WITH.MONITOR, 338
Process Property
AFTEREXIT, 318
BEFOREEXIT, 318
BUTTONEVENTFN, 303
WINDOWENTRYFN, 300
PROCESS.APPLY, 332
PROCESS.EVAL, 331
PROCESS.EVALV, 330
PROCESS.FINISHEDP, 321
PROCESS.RESULT, 325
PROCESS.RETURN, 324
PROCESS.STATUS.WINDO
W, 308, 310
PROCESSP, 319
PROCESSPROP, 315
PROCESSWORLD, 307

READGRAPH, 242
rectangle, 270
RECTANGLE, 271
RELDRAWTO, 260
RELEASE.MONITORLO
CK, 342
RELPROCESSP, 320
RESET/NODE/BORDER,
238
RESTART.PROCESS, 323
SET.TTYINEDIT.WINDO
W, 142
SPAWN.MOUSE, 303
structure editor, 88
SUSPEND.PROCESS, 327
TEdit, 16
AFTERQUITFN, 44
BACKSPACE key, 22
CARETLOOKSFN, 45
Character Looks Menu, 31
CHARLOOKS object, 62
Command Menu, 23
CTRL key, 20
DEL key, 22
editing pane, 17
ESC key, 22
Expanded Menu command,
29
Find command, 26
Format Specification
object, 63
Get command, 25
Include command, 25
Line Cache object, 61
Line Descriptor Object, 56

INDEX

- LOOKS, 44
 - Looks command, 28
 - Page Layout Menu, 35
 - Paragraph Looks Menu, 34
 - piece, 59
 - prompt pane, 17
 - Put command, 24
 - Quit command, 26
- QUITFN, 41
- SEL, 42
 - Selection Object*, 54
- SHIFT key, 21
- Substitute command, 27
- TEDIT.DEFAULT.FONT,
 - 40
- TEDIT.EXTEND.PENDIN
 - G.DELETE, 84
- TERMTABLE, 42
 - Text Object*, 47
- THISLINE object, 60
- TITLEMENUFN, 44
 - type-in point, 22
- UNDO key, 22
- TEDIT**, 38
 - TEDIT.COPY.LOOKS, 82
 - TEDIT.DEFAULT.FMTSPE
 - C, 84
 - TEDIT.DELETE**, 73
 - TEDIT.FIND, 74
 - TEDIT.GET.LOOKS, 81
 - TEDIT.GETSEL**, 69
 - TEDIT.HARDCOPY, 76
 - TEDIT.INSERT**, 71
 - TEDIT.LOOKS, 77
 - TEDIT.MOVESELECTION,
 - 86
 - TEDIT.QUIT, 84
 - TEDIT.READTABLE, 86
 - TEDIT.SELECTION, 85
 - TEDIT.SETSEL**, 67
 - TEDIT.SHIFTEDSELECTION,
 - N, 85
 - TEDIT.SHOWSEL**, 69
 - Text Editor, 16
 - Text Stream*, 46
 - TEXTSTREAM**, 66
 - THIS.PROCESS, 322
 - TTY.PROCESS**, 297
 - TTY.PROCESSP**, 298
 - TTYIN**, 114
 - ?ACTIVATEFLG, 145
 - DEFAULTPROMPT, 115
 - ECHOTOFILE, 126
 - TTYINAUTOCLOSEFLG,
 - , 144
 - TTYINCOMPLETEFLG,
 - 145
 - TTYINERRORSETFLG,
 - 145
 - TTYINREADMACROS,
 - 148
 - TTYINRESPONSES, 149
 - TYPEAHEADFLG, 114,
 - 144
 - TTYIN macro
 - ?=, 134
 - TTYIN Macro
 - BUF, 133
 - ED, 132

INDEX

EE, 133
FIX, 134
TV, 133
TTYIN.PRINTARGS, 137
TTYIN.SCRATCHFILE,
143
TTYINEDIT, 139

Type Size Menu, 28
UPPERRIGHT, 273
WAIT.FOR.TTY, 299
WAKE.PROCESS, 328
WBREAK, 157
WIDTH, 277