For
Macintosh Common Lisp
version 2

Getting Started With Macintosh Common Lisp

030-1958-A
Developer Technical Publications

Contents

Figures and tables

Preface About This Book

Contents

This Preface describes Macintosh Common Lisp version 2, including its new features and its documentation.

The Macintosh Common Lisp programming environment

Welcome to Macintosh Common Lisp version 2, a fluid and flexible environment for developing software tools and applications. Originally designed to handle the intricacies of artificial intelligence, Lisp is ideal for the complexities of programming the Macintosh computer.

Thirty years of evolution have made Lisp both efficient and rich in programmer-oriented features. Lisp code can be as fast as C or Pascal code, but Lisp handles automatically many programming issues such as memory management.

Macintosh Common Lisp implements Common Lisp and the Common Lisp Object System, as described in the second edition of *Common Lisp: The Language,* by Guy L. Steele, Jr., and others (Digital Press, 1990). (Note: this is the second edition, not the first, which describes an earlier stage of the language.) It provides partial implementation of the Metaobject Protocol as described in Gregor Kiczales and others, *The Art of the Metaobject Protocol* (MIT Press, 1991). Future updates of Macintosh Common Lisp will provide expanded Metaobject Protocol support.

Macintosh Common Lisp’s interactive programming environment saves time and simplifies debugging. You can compile, test, and debug functions individually, without having to recompile and relink an entire program.

Macintosh Common Lisp also provides a full range of tools and programming aids. These include a source-code stepper, a trace facility, a fast Lisp-oriented editor, and tools that give you quick access to the definitions of functions and variables.

Macintosh Common Lisp provides major portions of the Macintosh Toolbox as high-level Lisp objects. This design makes it easy to write programs that use the resources of the Macintosh interface. With Macintosh Common Lisp, it takes only minutes to create windows, menus, and dialog boxes. These tools make programming the Macintosh computer easy.

In addition, Macintosh Common Lisp provides complete low-level access to all Macintosh traps and data structures.

Read this manual first

This manual contains the information you need to begin using Macintosh Common Lisp on your Macintosh computer. *Experienced Macintosh Common Lisp users don't need to read all of this book.* Simply read Chapter 1, "Preparing to Run Macintosh Common Lisp," and the installation instructions in your MCL release notes.

If you have never used Macintosh Common Lisp before, you can take a quick tour of its features by reading Chapter 2, "A Tour of the System," Chapter 3, "Prototyping," and Chapter 4, "Debugging."

Here's what you'll find in this manual:

- Chapter 1, "Preparing to Run Macintosh Common Lisp"

  This chapter tells you what's in the product package and how much disk space and RAM you need to use Macintosh Common Lisp.

- Chapter 2, "A Tour of the System"

  This chapter introduces system features and shows you how to create, evaluate, edit, and save a simple program.

- Chapter 3, "Prototyping"

  This chapter shows you how to write a simple interface in Macintosh Common Lisp. Using the Interface Toolkit, a tool written in Macintosh Common Lisp, you create a dialog box and a menu, then attach the dialog box to the menu.

- Chapter 4, "Debugging"

  In this chapter, you debug code and recover from errors.

- Chapter 5, "Where to Go From Here"

  This chapter describes online and printed sources for further learning.

- Appendix A, "Fred Commands"

  This appendix lists standard commands in Fred, the Macintosh Common Lisp editor.

- Appendix B, "Common Errors and What to Do About Them"

  In this appendix you can read about common Macintosh Common Lisp errors and ways to recover from them.

- Appendix C, "Code Listings for Chapter 3"

  This appendix contains code examples used in Chapter 3.

For more information

Macintosh Common Lisp is shipped not only with this manual but also with the *Macintosh Common Lisp Reference* and release notes. You should also have access to the second edition of *Common Lisp: The Language,* by Guy L. Steele, Jr., and others. These reference manuals complement each other; *Common Lisp: The Language* documents Common Lisp, and *Macintosh Common Lisp Reference* guides you through the Macintosh Common Lisp extensions to Lisp. *Common Lisp: The Language* is available through APDA (see Chapter 5, "Where to Go From Here").

Macintosh Common Lisp's documentation and help features are probably your best source of information. As you get acquainted with Macintosh Common Lisp, you can use help balloons (autoloaded when you turn on Balloon Help) to understand what the menu items do. While programming, you can look at the documentation string of a function, view and edit the source code of many functions, and look through the whole namespace to find useful function names. In Chapter 4 of this manual, "Debugging," you'll learn how to use these as well as other useful debugging features.

Chapter 5 of this manual, "Where to Go From Here," lists more sources of information, including Macintosh Common Lisp's online technical bulletin board for developers. It also includes a bibliography of books on Common Lisp.

New features of Macintosh Common Lisp version 2

If you have used earlier versions of Macintosh Common Lisp, you will find many new features and changes in version 2.

- Macintosh Common Lisp is now compatible with the draft standard for Common Lisp as described in the second edition of *Common Lisp: The Language.*
- The Common Lisp Object System (CLOS) has replaced ObjectLisp, the object system previously used by Macintosh Common Lisp. All object functions have been rewritten as methods of generic functions. Objects have been replaced by classes and instances. Many older functions are now generic functions.

  Much of your code will continue to run unchanged. However, you must convert old ObjectLisp code to CLOS. In large part this is a mechanical process. See Appendix D of the *Macintosh Common Lisp Reference,* "Converting Your Files to CLOS."

- Macintosh Common Lisp now has a more powerful Inspector, within which values can be edited. The `trace` and `step` functions have been rewritten, and there is now an `advise` function. In general, debugging functions are more numerous and powerful. See Chapter 4, “Debugging,” in this manual and Chapter 10, “Debugging and Error Handling,” in the *Macintosh Common Lisp Reference.*
- The implementation of views, windows, and dialog boxes has changed. Many of the keywords previously handled at a lower level—by a window, dialog box, or dialog item—are now handled by views. Dialogs have virtually disappeared as a separate class. See Chapter 5, “Views and Windows,” and Chapter 6, “Dialog Items and Dialogs,” in the *Macintosh Common Lisp Reference.*
- Functions previously handled by dialogs are now handled by either Fred windows or Fred dialog items. See Chapter 14, “Programming the Editor,” in the *Macintosh Common Lisp Reference.*
- The implementation of menus has been simplified and changed. See Chapter 4, “Menus,” in the *Macintosh Common Lisp Reference.*
- The AID package of tools for designing interfaces has been rewritten, extended, and renamed. It is now called the Interface Toolkit. See Chapter 8, “The Interface Toolkit,” in the *Macintosh Common Lisp Reference,* as well as Chapter 3 of this manual.
- The file system has been changed. Old Macintosh Common Lisp logical pathnames have been renamed. Common Lisp logical hosts have been implemented. See Chapter 9, “File System Interface,” in the *Macintosh Common Lisp Reference.*
- The low-level interface has some new features and changes, including several incompatible changes. Because Macintosh Common Lisp now addresses a full 32 bits, Lisp objects may be stored in many more locations than was previously possible. For this reason, it is no longer feasible to tell the data type of an object from its address. Macintosh Common Lisp now includes a new Lisp data type, `macptr`, a pointer to a Macintosh data structure. You cannot pass any other Lisp object, including `nil`, to a function requiring a `macptr`. A `macptr` may never point to a Lisp object, only to the address of a Macintosh data structure. The Pascal null pointer is no longer equivalent to Macintosh Common Lisp `nil. A` null `macptr` can be created by the function `%null-ptr`. See Chapter 16, “Low-Level Operating System Interface,” in the *Macintosh Common Lisp Reference* for a full description of these changes.
- The traps functionality has been rewritten and made more robust. Traps interfaces now are autoloaded when required. The implementation of records is also more robust. See Chapter 17, “Higher-Level Operating System Interface,” in the *Macintosh Common Lisp Reference.*

- All external functions are now documented online.
- If you are running Macintosh system software version 7.0 or later, all menu items now have help balloons, which are autoloaded when you turn on Balloon Help.
- The documentation contains more examples and new features, including this book.

Chapter 1 Preparing to Run Macintosh Common Lisp

Contents

This chapter tells you what should be in your product package and what to do if anything is missing. It tells you what your system requirements are and what you should know about the Macintosh computer before you install Macintosh Common Lisp.

Installation instructions appear in the release notes included with your copy of Macintosh Common Lisp.

What’s in your product package

Your product package should include the following items:

- APDA and Development Tools information
- your product disks
- *Getting Started With Macintosh Common Lisp,* this document
- *Macintosh Common Lisp Reference*
- release notes

If you are missing any of these, please contact APDA. See Chapter 5 for how to do this.

You should also have access to the following book, which is the most current published description of Common Lisp with CLOS. It is available through APDA:

- Guy L. Steele, and others, *Common Lisp: The Language*, second edition (Digital Press, 1990).

System requirements

Macintosh Common Lisp version 2 requires approximately 4.5 MB of disk space. (The release notes give exact figures for both the basic system and options such as the Toolbox interfaces.).

Macintosh Common Lisp should be given a partition of at least 2 MB; 3 MB is recommended. Increasing partition size improves performance.

You may wish to turn off the RAM cache and remove memory-hungry `init` files before starting Macintosh Common Lisp.

Before you install

If you're new to the Macintosh world, you should become familiar with the basics of the Macintosh Operating System.

You should know how to do the following on the Macintosh computer:

- use the mouse to move the pointer, select an icon, and drag it
- choose a command from a menu
- create, activate, move, and close windows
- start an application
- name your work and save it on a disk
- work with folders and directories
- work with multiple windows
- insert, select, move, replace, and remove text using the Macintosh text editor

You can find explanations of how to do all of these in the *Macintosh System Software User's Guide* that came with your Macintosh computer. For practice, you can also try some Macintosh applications, such as MacWrite® and MacPaint®. Macintosh Common Lisp uses the same conventions as these applications do.

Installation

Install Macintosh Common Lisp, create a new folder called MCL 2, and follow the installation instructions in the release notes that come with your copy of Macintosh Common Lisp.

If you are upgrading from a previous version of Macintosh Common Lisp, your source files will probably require some upgrading to run under Common Lisp with CLOS. You may find it convenient to move your source files from your previous MCL folder to your version 2 folder at the time that you upgrade them.

Chapter 2 A Tour of the System

Contents

This chapter introduces the Macintosh Common Lisp environment, a powerful tool with which you can write and test programs. In this chapter you'll create a simple program and learn how to use the basic environments of Macintosh Common Lisp. You won't learn how to program in Lisp, of course, but you'll get a sense of the flexibility and ease of the Lisp environment.

In Chapter 3, you'll create a dialog box and a menu containing it, then connect a menu item to an external program. In Chapter 4, you'll learn how to debug errors in Macintosh Common Lisp. If you have previously used Macintosh Common Lisp, you don't need to read these chapters. However, make sure you are familiar with the many ways of getting help and debugging code in Macintosh Common Lisp, discussed in Chapter 4 of this book and in Chapter 10 of the *Macintosh Common Lisp Reference.*

If you're unfamiliar with CLOS, read "The Common Lisp Object System," Appendix C of the *Macintosh Common Lisp Reference*. If you're upgrading code from Macintosh Common Lisp 1.3, look at the material in Appendix D of the *Macintosh Common Lisp Reference,* "Converting Your Files to CLOS."

As you get more familiar with Macintosh Common Lisp, you can look at examples of source code in the Example and Interface Toolkit folders in your Macintosh Common Lisp version 2.0 folder.

The Macintosh Common Lisp interface is familiar

If you have experience with either the Macintosh computer or Lisp, you already know how to use Macintosh Common Lisp. Macintosh Common Lisp usually accommodates both the Macintosh way and the Lisp way of doing things. This is part of the richness and flexibility of the Macintosh Common Lisp environment.

If you have worked with any other implementation of Lisp, you will find that Macintosh Common Lisp feels very similar to most other Lisp implementations:

- You work in Common Lisp and use the Common Lisp Object System.
- You work with a Listener and an Emacs-like editor, in multiple windows.
- At any time you can look at and edit the definition of any function in your source code. You can look up a documentation string for a function.
- You can debug your source code in many ways, including stepping through it, tracing it, and inspecting it with an editable Inspector.
- Your environment is always fully programmable.

In addition, Macintosh Common Lisp conforms to the familiar conventions of Macintosh interfaces.

- If you are familiar with the Macintosh computer, you can use the editing style and procedures that you already know.
- If you're not familiar with the Macintosh Operating System, take time for a quick review. You should know how to use menus and the mouse, how to manipulate multiple windows, how to start an application, create files, and save them to disk, and how to use the Macintosh text editor.

  For explanations of how to do all of these, see the *Macintosh System Software User's Guide* that came with your Macintosh computer.

You can mix Macintosh with Lisp commands. And, as you'll see, you can easily modify your Macintosh environment in Lisp to make it do exactly what you want.

Getting into Macintosh Common Lisp

If you followed the installation instructions in the release notes that came with your copy of Macintosh Common Lisp, you now have a full version of Macintosh Common Lisp in a folder on your Macintosh desktop.

To start Macintosh Common Lisp, open this folder. Inside, you will see a set of other folders and the executable file `MCL 2.0`.

- **Move the mouse cursor over the icon next to `MCL 2.0` and click twice.**

This action loads the kernel, initializes the Lisp system, and loads the file `init.lisp` or `init.fasl` (if one of these is present in the same directory as Macintosh Common Lisp). This `init` file contains a listing of your environment preferences—any special behavior you want Macintosh Common Lisp to do or any settings you want to be in effect when it starts. (You can also set some preferences by choosing Environment from the Tools menu.)

After Macintosh Common Lisp starts, you see the Macintosh Common Lisp menu at the top of the screen. In the middle of the screen you see a Listener window containing a welcome message.

Macintosh Common Lisp commands

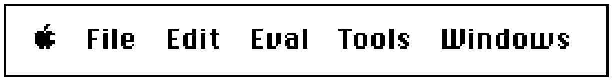
The Macintosh Common Lisp menu bar (Figure 2-1) contains six menus. The first three are familiar from other Macintosh applications:

- The Apple menu, indicated by an apple ( ), has two sections. Choose the first command to see information about Macintosh Common Lisp. The rest of the menu provides access to other Macintosh software such as desk accessories.
- Use the File menu to create and close editor windows, to print, and to quit Macintosh Common Lisp.
- Use the Edit menu to edit text.

The last three menus are specific to Macintosh Common Lisp.

- The Eval menu provides tools for evaluating expressions, loading and compiling files, and interrupting and continuing program operation.
- The Tools menu gives access to a variety of programming aids.
- The Windows menu lists all existing Macintosh Common Lisp windows.

n **Figure 2-1** The Macintosh Common Lisp menu bar



You choose a command from the Macintosh Common Lisp menu in the standard Macintosh way. Position the mouse cursor on the name of the menu in the menu bar (Figure 2-1), press the mouse button, hold it down while you drag the mouse cursor to the command you want, and then release the mouse button.

Sometimes a menu item is dimmed, indicating that you can't use it. For instance, when you first open the Listener, the Undo command in the Edit menu is dimmed. You can't choose this command when it's dimmed (you can't undo anything since you haven't edited anything yet).

Certain menu items are followed by names of keys. These command keystrokes are shortcuts for giving frequently used commands. For instance, the command to open a file is followed by -O. This means you can open a file in two ways: by choosing Open or by pressing the Command key and O at the same time.

The remainder of this section describes the menus in the menu bar.

Until you become familiar with MCL menu items, you may wish to turn on Balloon Help to remind yourself of what they do. Balloon Help is available in Macintosh system software version 7.0 or later. To turn on Balloon Help, move your mouse to the small balloon on the menu bar, press down, and move your mouse down until it is over the menu item Show Balloons. Then release the mouse button.

Apple command

The following command on the Apple menu applies to Macintosh Common Lisp:

About Macintosh Common Lisp
Displays a dialog box showing the version number of Macintosh Common Lisp and other copyright information.

File commands

The following commands control files:

New (-N)	Creates an editor window for a new file.
Open... (-O)	Allows you to select a text file and creates a new editor window for the file.
Open Selected File (-D)	If there's a selection in the top editor window, Macintosh Common Lisp attempts to parse this selection as a pathname and to create an editor window for the pathname's file.
Close (-W)	Closes the current window. If the window is not a Listener, and if its contents have been edited since the last time it was saved, Macintosh Common Lisp displays a dialog box asking you if you want to save or throw away the changes.
Save (-S)	Saves the contents of the active window to the file named in the title bar of the window. If the window isn't associated with a disk file, Save As is invoked.
Save As…	Allows you to specify a directory and filename, and saves the contents of the active window to that directory / filename.
Save Copy As…	Allows you to specify a directory and filename, and saves a copy of the contents of the active window to that directory / filename. Save Copy As is often used to save the contents of the Listener to a file.
Revert (-R)	Reverts to the version of the window contents last saved to disk. Before the reversion occurs, you're asked to verify whether you really want to revert to the last version saved.
Page Setup...	Allows you to set printing options for the current printer.
Print... (-P)	Prints the contents of the active window on the currently selected printer.
Quit (-Q)	Closes all windows and exits the Lisp environment. If any window contains revisions that have not been saved to disk, you're given the option of saving them.

Edit commands

The following commands control editing in the Macintosh Common Lisp environment. The file `font-menus.lisp`, in the Examples folder, shows how to supplement these with a set of font menus.

Undo (-Z) Undoes the last editor command, if possible.

The Undo command applies only to syntactic transforms, such as cutting, copying, pasting, etc., and not to all the commands that Fred knows how to perform. In general, Undo works for Cut, Copy, Paste, Clear, Select All, and Undo, but does not undo movement of the insertion point. When Undo will not work on the last command, Undo is dimmed.

Undo usually indicates what it can undo by changing its name. For example, if you have just cut text, Undo becomes Undo Cut; if you have just pasted text, Undo becomes Undo Paste.

Undo More (-M)
Undoes previous editor commands in order. The same restrictions apply as with Undo.

Cut (-X) Deletes the selected region and places it in the Clipboard and on the kill ring. (The kill ring is Macintosh Common Lisp's more powerful equivalent of the Clipboard.)

Copy (-C) Copies the selected region to the Clipboard and to the kill ring.

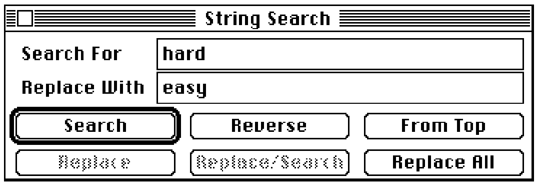
Paste (-V) Replaces the currently selected text with the text from the Clipboard. If no text is currently selected, the text is simply inserted at the insertion point.

Clear Deletes the currently selected text. The deleted text is not copied to the Clipboard. (However, like all deleted text, it *is* pushed onto the kill ring.)

Select All (-A) Selects the entire contents of the active window.

Search... (-F) Displays the String Search dialog box (Figure 2-2) with several options.

n **Figure 2-2** The String Search dialog box



Eval menu commands

The Eval menu contains the following eight commands:

Eval Selection (-E)

Evaluates the current selection in the top editor buffer. (The buffer is contained in the window; it is a copy of the file being edited.) If there is no selection and the insertion point is next to a parenthesis, the expression bounded by the parentheses is evaluated.

Eval Buffer (-H)

Evaluates the entire contents of the top editor buffer. This is equivalent to choosing Select All followed by Eval Selection.

Load... (-Y) Allows you to select a file for loading into Macintosh Common Lisp. You may load both `lisp` and `fasl` files (respectively, uncompiled and compiled Lisp files).

Compile File... Allows you to select a file for compilation. You are asked to specify both the source and destination files.

Abort (-period)

Cancels the current computation and returns to the read-eval-print loop (the functionality through which Macintosh Common Lisp reads an expression, evaluates it, and prints the result). If Macintosh Common Lisp was in a break loop, it leaves the loop. A canceled computation cannot be resumed. You'll learn more about this command in the section "Recovering From an Error With Abort," in Chapter 4.

Break (-comma)

Suspends the current computation and enters a break loop. The state of the machine can be examined in the break loop. You can resume the computation by choosing Continue or by calling the function `continue`. You'll learn more about this command in the section "The Break Loop," in Chapter 4.

Continue (-slash)

Continues the last operation halted by a break or by a continuable error. (To give the command keystroke, press the Command key and the / key.) This command is disabled when it is not applicable. You'll learn more about this command in the section "Recovering From an Error With Continue," in Chapter 4.

Restarts... (-backslash)

Provides a list of possible ways to restart the current operation. (To give the command keystroke, press the Command key and the \ key.) This command is disabled when it is not applicable. You'll learn more about this command in the section "Recovering From an Error With Restarts," in Chapter 4.

Tools menu commands

The Tools menu contains these 11 commands:

Apropos Displays the Apropos dialog box. This box accepts a string entered by the user and displays a table of symbols that match that string. For example, if you type the string `view`, Apropos displays a table listing `view`, `simple-view`, `view-draw-contents`, and many more expressions containing `view`.

Documentation...

Gets a symbol name from the user and displays the documentation string for the symbol, if a documentation string is available.

Edit Definition...

Allows the user to enter a symbol name and attempts to find the source code for the definition of the symbol.

Inspect Displays the Inspector Central window.

List Definitions Displays a modeless dialog box containing a table of all the definitions in the editor buffer displayed in the frontmost window. The user can select a definition, and the buffer will scroll to that definition.

This command is disabled when the active window is not an editor window.

Search Files Displays a dialog box for searching a set of files for a given string. The user can use wildcard characters to specify the set of files to search. (Macintosh Common Lisp supports Common Lisp extended wildcards, which are documented in *Common Lisp: The Language;* see also the section "Wildcards" in Chapter 9, "File System Interface," of the *Macintosh Common Lisp Reference.*)

Backtrace (-B)

This command is enabled only when Macintosh Common Lisp is in a break loop. Choosing it brings up the Stack Backtrace window, described in Chapter 4, "Debugging."

Fred Commands Displays a window listing all the commands available in Fred, the Macintosh Common Lisp editor. You can search through this list using the Fred search command, Control-S, and can copy text from it to another Fred window. See the section "Getting Help on Fred Commands," later in this chapter.

Listener Commands

Brings up a window listing all Listener commands that differ from Fred commands.

Print Options... Brings up a dialog box that allows the user to set the values of several Lisp printer parameters. These are described in Chapter 1, "The Macintosh Common Lisp Environment," in the *Macintosh Common Lisp Reference.*

Environment... Brings up a dialog box that allows the user to set the values of several global variables affecting the Macintosh Common Lisp environment. These are described in Chapter 1, "The Macintosh Common Lisp Environment," in the *Macintosh Common Lisp Reference.*

Windows menu

The titles of all visible windows appear as items in this menu. The menu is ordered by window layer. (That is, the front window—the window currently in use—is the first menu item and the back window is the last menu item.)

To activate a window, choose its menu item. You can also select the Listener by giving the command keystroke Command-L.

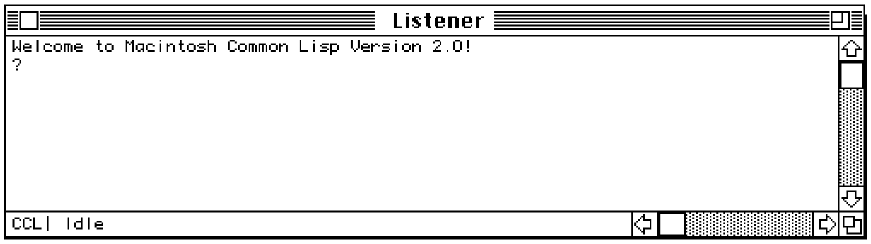
You can make the front window the back window by holding down the Option key and clicking the window's title bar.

In the menu, windows whose contents have been changed but not saved are marked with a cross next to their names. The name of the currently selected window is dimmed and cannot be chosen.

Using the Listener

When you first open Macintosh Common Lisp, you see a single window. This is the Lisp Listener (Figure 2-3), the first of the two basic windows available to you.

n **Figure 2-3** A Listener window



```
Welcome to Macintosh Common Lisp Version 2.0!
?
```

```
CCL| Idle
```

The Lisp Listener (or just Listener for short) is the special window designed for interactive Lisp programming. Most interaction between the programmer and Lisp takes place in this special window. As soon as you type a complete piece of code in the Listener, Lisp reads what you've typed, evaluates it, and prints the result. This interaction facility is called the **read-eval-print loop.** It's characteristic of the Listener.

You'll recognize the Listener from its name in the title bar, "Listener," and from the question mark prompt. Whenever you see this question mark, the Listener is ready to read input.

The Listener is natural and straightforward to use. Text you enter into the Listener appears in **`boldface.`** Macintosh Common Lisp's response appears in `Normal type`.

When you see examples in the text, the bold text indicates what you type. The plain text shows what Macintosh Common Lisp prints in response.

```
? "Hello, world!"              <—This is what you type.
"Hello, world!"                <—This is Macintosh Common Lisp's response.
```

Evaluating expressions in the Listener

When you type an expression and press Return, Macintosh Common Lisp immediately evaluates the expression—finds its value—and returns a result. You can see an example by getting the Listener to evaluate a string.

Among the simplest Lisp expressions are strings. A **string** is a one-dimensional array whose elements are characters. In Lisp, the beginning and end of a string are indicated by double quotation marks.

Here are some examples of strings:

```
"dog"
"cat"
"XJ313"
```

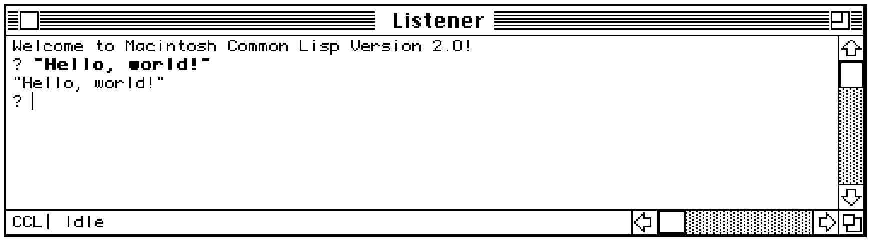
The value of a string in Lisp is the string itself:

```
? "dog"
"dog"
? "cat"
"cat"
? "XJ313"
"XJ313"
```

Here's a simple evaluation:

- **Type `"Hello, world!"` and then press Return.**

n **Figure 2-4** A simple evaluation in a Listener window



```
Welcome to Macintosh Common Lisp Version 2.0!
? "Hello, world!"
"Hello, world!"
? |
```

As Figure 2-4 shows, this evaluation returns the value `"Hello, world!"`

The Listener is a scratchpad showing calculations within a session

The Listener is like a scratchpad or the display window of a calculator. You enter information into it, and it displays that information. As soon as you complete a step of your code, Macintosh Common Lisp evaluates it and prints the result in the Listener.

Like a calculator, the Listener has memory. If you want to reuse a previous entry without retyping the code, press Option-G.

n **Press Option-G to retrieve the string `"Hello, world!"`.**

Pressing Return evaluates the string again. Pressing Control-G cancels the evaluation.

To insert a carriage return without evaluating, press Command-Shift-Return.

The Listener also acts like a piece of paper because it preserves your old calculations. Until the Listener window is closed, you can look at the history of your current session by scrolling the window upward.

You can copy any useful code into a file, or save the whole Listener session as a file by choosing Save Copy As from the File menu.

At any time you can “tear off” the current sheet of scratchpaper, and get rid of your visible calculations, by closing the Listener window. Macintosh Common Lisp automatically creates a fresh Listener, where you can continue programming. Information in memory—the result of your calculations—is preserved. The only thing you have thrown away is the visible session history.

This interaction shows that you can preserve information across Listener windows:

1. **Set a symbol to a value.**

   ```
   ? (setf x 5)
   5
   ```

2. **Close the Listener window.**

   Click the close box in the upper-left corner of the Listener.

   The Listener window closes.

   Macintosh Common Lisp automatically creates a fresh Listener window.

3. **Evaluate the symbol `x` in the new Listener.**

   Type `x` in the new Listener.

   ```
   ? x
   5
   ```

But the Listener doesn't preserve information from session to session; it simply reads code, evaluates it interactively, and prints a result.

If you set a symbol to a value in the Listener—for instance, if you give `x` the value 5—Macintosh Common Lisp preserves that information until the end of the current session. But, if you were to close Macintosh Common Lisp and open it again, `x` would no longer have a value:

```
? x
> Error: Unbound variable: X
> While executing: SYMBOL-VALUE
```

(You can cancel this error—"abort" it—by pressing Command-period or choosing Abort from the Eval menu.)

Preserving work from session to session

To preserve work from session to session, you must save it as source code. All source code in Lisp is simply text files, which you can edit with the Macintosh Common Lisp editor, Fred, described later in this chapter in "The MCL Editor, Fred."

You can save a copy of the Listener to a file that you can edit in Fred, using Save Copy As from the File menu. However, you will more often work directly from Fred and evaluate your code in the Listener.

When you want to use the source code, you simply open the source file and evaluate the code. For example, if you wish to have the value of `x` set to 5 in a subsequent session, save the expression `(setf x 5)` in a file. At the beginning of the subsequent session, open the file and evaluate the expression.

The process of opening a file and evaluating its code is called **loading** the file. To load a file, choose Load from the Eval menu.

Comparing the Listener and the editor

Here is a summary of the differences between what the Listener does and what the editor does:

- The Listener window reads input interactively through the read-eval-print loop. It doesn't save information from session to session; to do that, you copy Listener information to a file, which you can then edit.
- In an editor window, you edit files. You can save information in these editable files from session to session. Within the editor, you can evaluate code, but the results of evaluations always appear in the Listener.

The MCL editor, Fred

Macintosh Common Lisp includes a powerful editor, Fred, based on the standard Lisp editor Emacs. Fred ("Fred Resembles Emacs Deliberately") includes the usual Macintosh editor features, such as multiple windows and mouse-based editing. If you have edited text on the Macintosh before, you can probably get started with Fred without reading any documentation.

Fred also includes a sophisticated set of features designed specifically for editing Lisp. Among these are parenthesis matching, smart indentation, and Lisp-oriented keyboard commands.

Moreover, because Fred is written in Macintosh Common Lisp, it is fully extensible: you can program it to include new commands.

Fred and the Listener work in tandem.

- You usually develop code in a Fred window (which is perhaps a copy of a Listener session).
- You see the results of evaluating code in the Listener.
- Almost all of the Fred editing features work the same way in Fred windows and in the Listener.

To get into Fred, simply open a file or create a new file.

In the next few sections, you will learn how to do the following in a Fred window:

- create a new window
- add text
- select text
- evaluate expressions
- copy text
- save work to a file on disk

You'll create a dialog box that asks your name and prints `"Hello,` *your-name*`!"`

Creating a Fred window

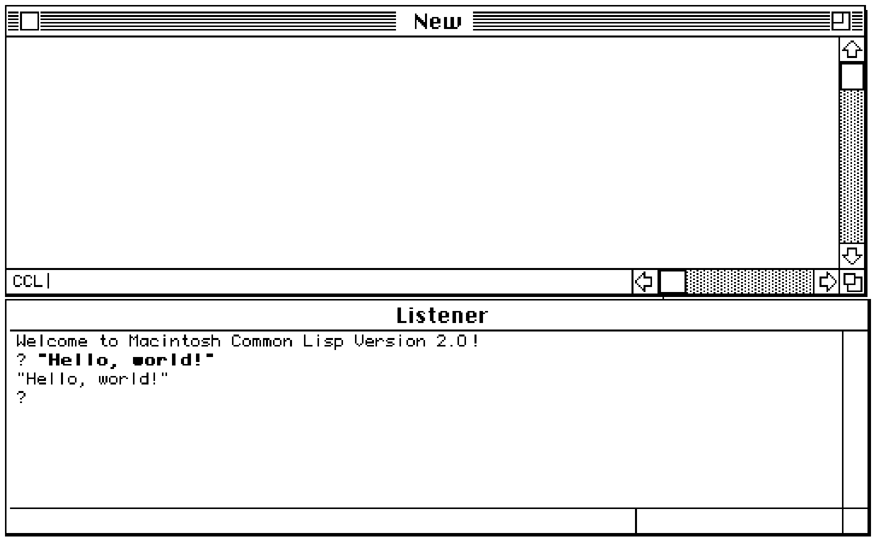
Fred windows are created whenever you open a new or an existing file:

- **Open a new file.**

  Choose New from the File menu.

  You see two windows, your original Listener and a new Fred window, as in Figure 2-5. (Their onscreen positions may be different from those in the figure.) The new Fred window becomes the active window.

n **Figure 2-5** A Fred window and a Listener



```
Welcome to Macintosh Common Lisp Version 2.0!
? "Hello, world!"
"Hello, world!"
?
```

A Fred window displays an editor buffer, a copy of the file you're editing. If the file is new, it doesn't yet exist on your disk; the only copy is in the buffer. If you're editing a file that already exists, the file is on your disk and a copy is in the buffer. When you save the contents of the buffer, they are copied to the file. If you save the buffer with Save As and the file already exists, Fred asks you whether you want to overwrite the previous contents of the file.

Adding text to a Fred window buffer

Lisp saves code as simple text files, so you edit them as simple text. To add code in a Fred buffer, simply type some text.

n **Type `"Hello, world!"` and press the Return key.**

Fred works differently than the Listener does. When you type a string in a Listener window and press Return, the Listener evaluates the string. Fred doesn't automatically evaluate: text is just text and Return is just Return.

However, you can easily evaluate the expression you have just typed.

Evaluating in a Fred window

When you are working in a Fred window, you can evaluate expressions by positioning your insertion point just after the expression and pressing Enter (not the Return key, but the Enter key on the numeric keypad) or Command-E.

You can also make a selection and then evaluate it:

1. **Select the string `"Hello, world!"`.**

   The entire expression is highlighted.

2. **Evaluate the expression by pressing Command-E or Enter.**

The results of an evaluation always appear in the Listener, where you should see `"Hello, world!"`

Counting parentheses and highlighting an expression

Traditionally, one of the problems in programming Lisp is balancing parentheses. Fred shows you where an expression begins and ends by causing one of a pair of parentheses to blink when you place the insertion point by the other.

1. **Press Return and type the following code, which asks your name:**

   ```
   (get-string-from-user
        "Please type in your name.")
   ```

   The opening parenthesis of the expression blinks when the insertion point is just after the closing parenthesis.

   When you double-click just after one parenthesis of a pair, Fred selects everything between it and the other parenthesis.

2. **Position the insertion point just after the closing parenthesis and double-click.**

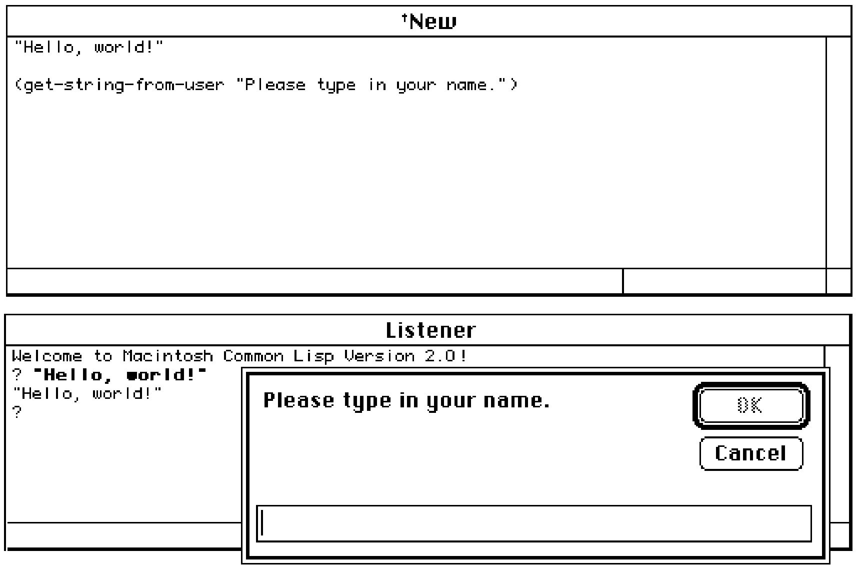
   The entire expression is highlighted. If this were an expression within an expression, only the part of the expression back to the corresponding open parenthesis would be highlighted.

   Now evaluate it.

3. **Press Command-E or Enter.**

   Macintosh Common Lisp evaluates the expression. Figure 2-6 shows the result.

n **Figure 2-6** Fred evaluates an expression



```
"Hello, world!"

(get-string-from-user "Please type in your name.")
```

```
Welcome to Macintosh Common Lisp Version 2.0!
? "Hello, world!"
"Hello, world!"
?
```

Please type in your name.

The function `get-string-from-user` creates a turnkey dialog box, which asks the user for a string.

4. **Type your name in the highlighted box and press Return.**

   Macintosh Common Lisp returns your name as a string in the Listener window.

(You can read more about turnkey dialog boxes in Chapter 6, “Dialog Items and Dialogs,” of the *Macintosh Common Lisp Reference*.)

Moving around in a Fred window

The file in your Fred window should now look like this:

```
"Hello, world!"
(get-string-from-user
          "Please type in your name.")
```

By simple editing, you can turn this into a more interesting program that gets your name and returns `"Hello,` *your-name*`!"`

1. **Create a symbol to represent the value of your name.**

   To do this, edit `get-string-from-user`.

   Move the mouse cursor to just before the opening parenthesis of `get-string-from-user` and click once to position the insertion point there.

   Type exactly the following, followed by a space:

   ```
   (defvar your-name)
   
   (setf your-name
   ```

   Position the mouse cursor just beyond the closing parenthesis of the `get-string-from-user` expression and click again. Type a second closing parenthesis there.

   This is what you should see:

   ```
   (setf your-name (get-string-from-user
        "Please type in your name."))
   ```

   You are asking Macintosh Common Lisp to perform a `get-string-from-user`, which returns your name, then to set the symbol `your-name` to that value.

   The use of the `defvar` macro declares that the variable `your-name` is a special variable in this program.

2. **Evaluate the expression.**

   Double-click just after the second closing parenthesis and press Enter or Command-E. The dialog box will appear and, after you type your name, it will be printed in the Listener.

Cutting and pasting text in a Fred window

Now edit `"Hello, world!"` so it becomes `"Hello,` *your-name*`!"` To practice moving text around, move `"Hello, world!"` so that it follows the code you have just edited.

This time try a Fred command. The command to delete a line is Control-K (not Command-K).

1. **Delete the line that contains the string `"Hello, world!"`**

   Position the mouse cursor at the start of the line and click to move the insertion point there.

   Press Control-K by holding down the Control key as you press K.

   The line is deleted from the copy of the file in your Fred window and added to Fred's kill ring. The Fred kill ring is similar to the Macintosh Clipboard but can hold many deletions instead of only one.

2. **Move to the bottom of the file.**

   If necessary, press Return once or twice to get a new line.

3. **"Yank" the deletion from the kill ring.**

   The Fred command for this is Control-Y.

The `format` function

You use the Common Lisp function `format` to create formatted output. You will type `(format t "Hello, ~A!" your-name)`. The parameter `t` makes `format` print in the Listener, and ~A specifies a particular type of format.

1. **Move to the beginning of the string `"Hello, world!"`.**
2. **Type `(format t` and a space.**
3. **Now replace `world` with `~A`.**
4. **Add `your-name` and the closing parenthesis to finish the new expression.**

Evaluating a buffer

You now have a file with three expressions in it, one of which is actually composed of two functions, `setf` and `get-string-from-user`. It should look like this:

```
(defvar your-name)

(setf your-name (get-string-from-user
          "Please type in your name."))

(format t "Hello, ~A!" your-name)
```

Now get the result of these expressions:

n **Evaluate the buffer.**

Choose Eval Buffer from the Eval menu.

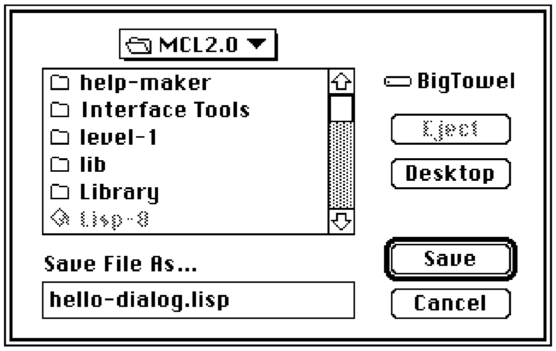
Saving a buffer to a file

To preserve code, you need to save it to a file.

1. **Choose Save from the File menu or press Command-S.**

   If you are working with a new file, Macintosh Common Lisp prompts you for a name, using the dialog box shown in Figure 2-7.

n **Figure 2-7** The Save File dialog box



Save File As...

hello-dialog.lisp

2. **Type the name `hello-dialog.lisp`. To accept the name, either press Return or click the Save button.**

Moving between a Fred window and the Listener

It is easy to move between windows in Macintosh Common Lisp. Simply click anywhere in the window where you would like to go.

Try it now. To check that you have set `your-name` to your name, go to the Listener and evaluate `your-name`.

1. **Move to the Listener window.**

   Click anywhere in the Listener window. The insertion point should be at the question mark prompt.

2. **Type `your-name` and press Return.**

   Macintosh Common Lisp evaluates `your-name` and returns its value.

Of course, you can also evaluate `your-name` within a Fred window by selecting `your-name` and pressing Command-E.

Getting help on Fred commands

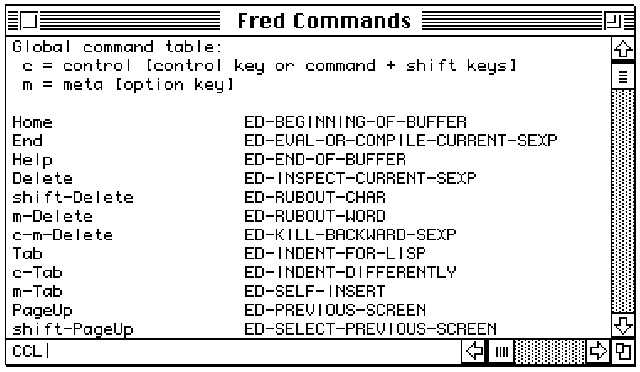
The Fred Commands window (Figure 2-8) lists all the commands currently available in Fred and their command keystrokes. It can be invoked by pressing Control-/ or Control-?.

If you know the name of the command you want, you can search for its command keystroke in this window, using the Fred search command, Control-S.

Here are some useful strings to search on when you don’t know the name of the command:

- `EVAL` (evaluate)
- `SEXP` (symbolic expression; Lisp expression)
- `KILL` (add to the kill ring)
- `CHAR` (character)
- `SCREEN`
- `REGION`
- `WINDOW`
- `BUFFER`
- `WORD`
- `COMPILE`
- `DELETE`
- `INSPECT`
- `MACROEXPAND`

n **Figure 2-8** The Fred Commands window



```
Global command table:
 c = control [control key or command + shift keys]
 m = meta [option key]

Home                    ED-BEGINNING-OF-BUFFER
End                     ED-EVAL-OR-COMPILE-CURRENT-SEXP
Help                    ED-END-OF-BUFFER
Delete                  ED-INSPECT-CURRENT-SEXP
shift-Delete            ED-RUBOUT-CHAR
m-Delete                ED-RUBOUT-WORD
c-m-Delete              ED-KILL-BACKWARD-SEXP
Tab                     ED-INDENT-FOR-LISP
c-Tab                   ED-INDENT-DIFFERENTLY
m-Tab                   ED-SELF-INSERT
PageUp                  ED-PREVIOUS-SCREEN
shift-PageUp            ED-SELECT-PREVIOUS-SCREEN
CCL|
```

For example, suppose that you want to find all the Fred commands that deal with expressions.

1. **Choose Fred Commands from the Tools menu.**

   The Fred Commands window appears. Its default size is rather small, and it is located at the bottom of the screen. You can change its size and position just as you would those of any other Macintosh window.

2. **Search for the string `sexp`.**

   Press Control-S, then type `sexp`. Macintosh Common Lisp matches the string as you type it. The search is not case-sensitive.

   The first occurrence of `sexp` is highlighted. (It should be in `ED-EVAL-OR-COMPILE-CURRENT-SEXP`.)

3. **Find its command keystroke.**

   The command keystroke is indicated in the first column of the Fred Commands window. C stands for the Control key; M stands for the Meta key (by default, the Option key). The command keystroke of `ED-EVAL-OR-COMPILE-CURRENT-SEXP` is the Enter key.

4. **Press Control-S again to continue searching.**

Copying information with Fred

You can copy useful Fred commands to a Fred window.

Here's how to copy from one window to another using Fred:

1. **Go to the beginning of the region you want to copy.**

   Use the mouse to position the insertion point.

2. **Indicate one end of the region you want to copy.**

   Press Control–Space bar to set a mark.

3. **Go to the end of the region.**

   Use the mouse to move there. Click to position the insertion point at the end of the region.

4. **Copy the information to the kill ring.**

   Press Option-W to copy the information in the region. (Option-W is the Fred Copy command keystroke.)

5. **Open a Fred file.**

   Choose New from the File menu, or open an existing file and select where you would like the copied information to go.

6. **"Yank" the information from the kill ring by pressing Control-Y, the Fred Paste command keystroke.**

Since Fred produces plain ASCII text, you can also import files from Fred into other environments such as word processors and databases.

You can read more about Fred in Appendix A of this book, "Fred Commands," and in Chapters 2 and 14 of the *Macintosh Common Lisp Reference.*

Getting help on Listener commands

A few commands work only in the Listener. The Listener Commands window lists all the Listener commands that differ from Fred commands, and their command keystrokes.

What you've learned about the Macintosh Common Lisp environment

You've learned that there are two basic windows in the Macintosh Common Lisp environment, the Listener and the editor. First you created a Listener window and evaluated code in it. Then you added a Fred window and edited a buffer (an editable copy of a file). You created a small program in the buffer and saved the buffer to a file. You've used some Fred commands and found out how to get help on Fred commands.

Remember, you can take a break at any time and later pick up where you left off. To get out of Macintosh Common Lisp, simply choose Quit from the File menu.

When you go on, you'll learn how to compile files, load and use the Interface Toolkit, do quick prototypes of your own dialogs and menus, and debug coding errors.

Chapter 3 Prototyping

Contents

Because Macintosh Common Lisp provides major portions of the Macintosh Toolbox as high-level Lisp objects, it is ideal for programming the Macintosh computer.

It is also outstanding for writing tools to create interfaces. The Interface Toolkit source code, an MCL application included in your files, shows you a typical interface tool created with Macintosh Common Lisp. In the next few minutes you'll use the Toolkit to create some user interface dialog boxes and a new menu.

You'll learn how to compile files for your own tools, load a tool, and create simple interfaces.

When doing your own work, you can extend the Interface Toolkit and create your own similar tools.

Compiling files

In the last chapter we introduced the read-eval-print loop. When you type an expression to the Listener, Lisp actually does more than evaluate. Macintosh Common Lisp uses an incremental compiler. When you evaluate an expression, Macintosh Common Lisp compiles it and runs the compiled code. Because the compiler is fast, there is no noticeable delay. During code development, you don't need to compile files repeatedly.

Lisp code is saved as simple text in editable files. That is, the text of the definitions and other expressions is saved. The compiled forms of the expressions are not saved. When you restart Lisp and open the file, you need to recompile the functions in the file before you can use them. Recompiling can take some time when you are working on a large project.

To avoid the need to recompile files every time you restart Lisp, Macintosh Common Lisp provides a **file compiler.** The file compiler takes a source code file (that is, a text file of Lisp expressions), compiles the file, and saves the compiled version in another disk file. This compiled file, called a **`fasl` file,** can be loaded quickly into Lisp.

You can compile files by choosing Compile File from the Eval menu. When you choose this command, Macintosh Common Lisp displays a directory dialog box, allowing you to pick a text file to compile. When you have selected a file, you are prompted for a name under which to save the file. The default is the name of the original text file, but with a `.fasl` extension.

You can also compile files by using the Common Lisp function `compile-file`.

𝕊 **Important** In Lisp you can compile files individually, without completely recompiling and relinking an entire program, just as you can compile individual expressions. 𝕊

Here is how to compile an individual file. You'll practice by compiling `hello-dialog.lisp`.

1. **Choose Compile File from the Eval menu.**
2. **Select `hello-dialog.lisp` from the list of files.**

   This is the file that you created in Chapter 2.
3. **Press Return or click the Compile button.**

You're more likely to use `compile-file` to compile many files at once. For an example of how to load and compile multiple files, look at the function `load-ift` in the file `make-ift.lisp`, which is in the Interface Tools folder.

The Interface Toolkit

This section shows you how to load the Interface Toolkit, a sample application written in Macintosh Common Lisp. The Interface Toolkit source code is supplied with Macintosh Common Lisp in the Interface Tools folder. Feel free to use and extend it.

The Interface Toolkit helps you create interfaces.

- It creates dialog windows. You can create a blank dialog window with any set of attributes you want. Then, from a palette of radio buttons, checkboxes, and static text, you can add dialog items by dragging them.
- It creates new menus and menu bars and populates them with menu items. You can use the Menubar Editor, part of the Interface Toolkit, to edit the default menu bar or another menu bar, giving it menu items and a set of attributes. You can add your own menu items to a menu bar, simplify a menu bar, or create a new menu bar.
- You can edit menus, dialogs, and items by double-clicking them and specifying their attributes in a Fred window.
- At any time you can generate source code for any menu or dialog created with the Interface Toolkit.

Loading the Interface Toolkit files expands the functionality of Macintosh Common Lisp. Bootstrapping on these tools, you can add new dialog boxes and new menus to Macintosh Common Lisp.

The Interface Toolkit is one example of the many applications and application-building tools you can create with Macintosh Common Lisp.

Loading the Interface Toolkit

To load the Interface Toolkit:

1. **Open the file `make-ift.lisp` in the Interface Tools folder and evaluate its contents.**

   Choose Open from the File menu. Find the Interface Tools folder and double-click to open it.

   Double-click the file `make-ift.lisp` to open it.

   Evaluate its contents by choosing Eval Buffer from the Eval menu.

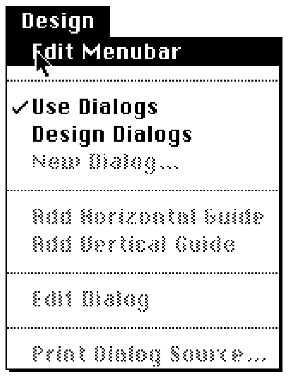
2. **Type the following in the Listener or evaluate it in a Fred window:**

   ```
   (ift::load-ift)
   ```

   This function loads the compiled and uncompiled files that make up the Interface Toolkit, compiles any uncompiled files, and runs them.

   A new menu, Design, appears on the Macintosh Common Lisp menu bar, as shown in Figure 3-1.

n **Figure 3-1** Loading the Interface Toolkit adds a Design menu to the menu bar



Creating a dialog box

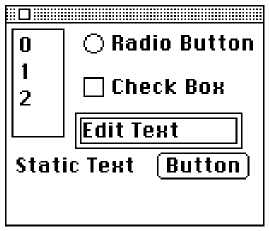
In this section you use the Interface Toolkit to create a new dialog box. This time you won't use a turnkey dialog box like `get-string-from-user`; you'll make your own. As a first step, make your dialog box run the code you've already developed, asking for your name and printing a greeting in the Listener.

1. **Open the Dialog Designer in the Interface Toolkit.**

   Choose Design Dialogs from the Design menu.

   You see a palette of dialog items (Figure 3-2).

n **Figure 3-2** The Design Dialogs palette of dialog items

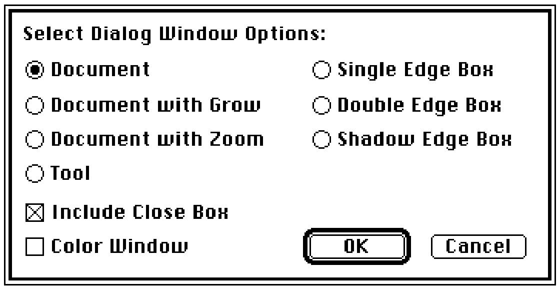


Now create a new dialog box.

2. **Select New Dialog from the Design menu.**

   You see a dialog box like the one in Figure 3-3.

n **Figure 3-3** New Dialog dialog box



Select Dialog Window Options:

(● Document) ○ Single Edge Box
○ Document with Grow ○ Double Edge Box
○ Document with Zoom ○ Shadow Edge Box
○ Tool

[✓] Include Close Box
[ ] Color Window

3. **Click the Document radio button and select the Include Close Box option if it is not already selected. Then click OK or press Return.**

   An empty dialog box, “Untitled Dialog,” appears on the screen.

   Now you can add items to the dialog box.

4. **Select the Static Text icon from the Design Dialogs palette and copy a static text item to the new dialog box.**

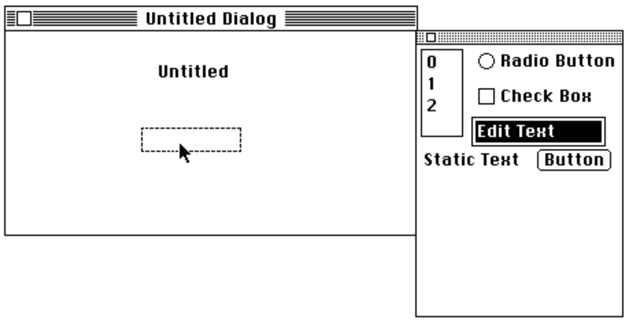
   Move the pointer over Static Text and press the mouse button. A box appears around the words Static Text.

   Drag the box to the upper center of the new dialog box.

5. **Select the Button icon from the palette and drag a button into the new dialog box.**

   Position the new button underneath the static text and move it until it is roughly in the position shown in Figure 3-4. In Figure 3-4, the user has pressed and dragged the Button icon. A box indicates the new button’s size and location.

n **Figure 3-4** Moving a new button into an untitled dialog box



Editing the dialog box interface

To customize a dialog box, edit it within the Dialog Designer. You can add new dialog items and specify actions corresponding to those items.

As you do this, note that the look and feel of the Interface Toolkit are very similar to those of the rest of Macintosh Common Lisp; for instance, the editor the Dialog Designer uses behaves just as Fred does. This is more than good design (although it helps you design well). It's actually the same code: Interface Toolkit dialog boxes use the same MCL classes and methods to determine their editing behavior as Fred windows do.

Because Macintosh Common Lisp is modular and object-oriented, it's easy for your programs to maintain a coherent look and feel. If you want to change the way the system operates, you can do it coherently and thoughtfully, at whatever level you please, by changing the behavior of the appropriate MCL classes.

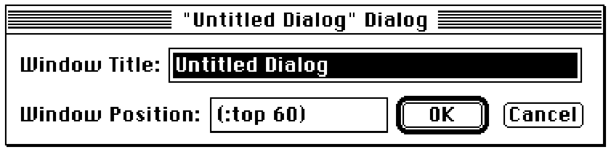
Editing dialog boxes and dialog items

To edit the dialog box, follow these steps:

1. **Choose Edit Dialog from the Design menu.**

   You see a dialog box like the one in Figure 3-5, indicating everything you can edit in your own dialog box.

n **Figure 3-5** Edit Dialog dialog box


"Untitled Dialog" Dialog

Window Title: Untitled Dialog

Window Position: (:top 60)

2. **Edit the title by typing the following over the highlighted text:**

   `Good morning!`

3. **Click the OK button to close the Edit Dialog dialog box.**

   The new title appears at the top of the dialog box, replacing "Untitled Dialog."

Now you can edit the items in the dialog box. To edit any item, double-click it.

1. **Edit the static text.**

   Double-click the Static Text item.

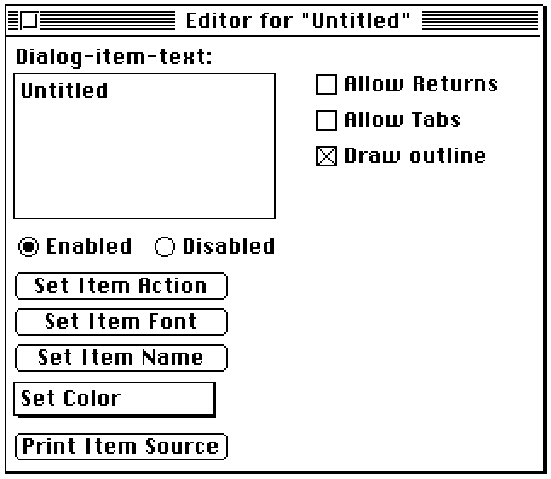
   Replace `Untitled` with the following text:

   ```
   Press the button to see the Listener greet you.
   ```

   You do not need to press Return after typing this. (Pressing Return will only add a blank line to the text.)

   The text will not naturally wrap in the Edit Dialog Items dialog box. (In a moment, though, you'll make it wrap in the dialog box you're creating.)

n **Figure 3-6** Edit Dialog Items dialog box



2. **Close the Edit Dialog Items dialog box to return to your new dialog box.**

3. **Change the display area of the static text.**

   Click the Static Text item. Handles appear at the boundaries of the area covered by the item. Change the size of the area by positioning the pointer over a handle, holding down the mouse button, and dragging the mouse.

   When the area looks right to you, release the mouse button.

4. **Edit the button.**

   Double-click the button. An Edit Dialog Items dialog box opens, similar to the one in Figure 3-6.

   Give the button the title `Press here` by editing `Untitled.`

   Display the button's title in Monaco 12 by clicking the Set Item Font button and selecting Monaco 12.

   Make the button the default button by clicking the checkbox next to Default Button.

   Close the Edit Dialog Items dialog box and return to the dialog box you're creating.

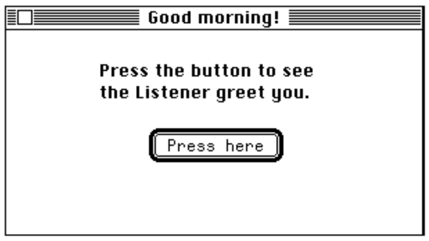
   Change the size of the button by clicking the button and dragging its handles.

5. **Move your dialog items until their positions satisfy you.**

   You move dialog items by positioning the pointer over them, holding down the mouse button, and dragging them.

   When you are done, your dialog box should look like the one in Figure 3-7.

n **Figure 3-7** Size and move your items until they satisfy you



Adding item actions

Dialog boxes exist to perform some action. In this section you learn how to associate a dialog item with an action. To do so, you'll use an already debugged piece of code.

1. **Copy your code from `hello-dialog.lisp` to the Fred kill ring.**

   Open the file `hello-dialog.lisp`, which you created in Chapter 2.

   Place the insertion point at the end of `(defvar your-name)` and press Command-E to evaluate that expression.

   Select the remainder of the file contents by holding down the mouse button and dragging to the end of the buffer.

   Press Option-W, the Fred command to copy a selection into the kill ring.

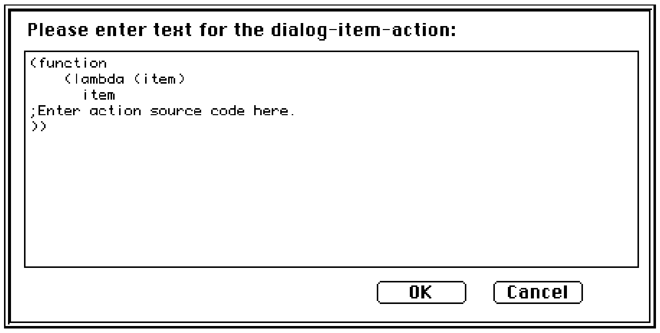
   Now close the window containing `hello-dialog.lisp`. You won't need it again.

2. **Double-click the "Press here" button to activate its edit window.**

3. **Click the button Set Item Action.**

   You see a dialog box containing a window of editable text, like the one in Figure 3-8.

n **Figure 3-8** Dialog item action window containing Lisp source code



Please enter text for the dialog-item-action:

```
(function
    (lambda (item)
      item
;Enter action source code here.
))
```

4. **Select the fourth line and replace it with your code.**

   Use the Fred “yank” command, Control-Y.

   Your code replaces the line that says `;Enter action source code here`. Make sure you replace the leading semicolon as well as the rest of the line. If you don’t, Macintosh Common Lisp reads the first line of your code as a comment.

   Your changed code should look like this. The change is in boldface:

```
(function
      (lambda (item)
       item
(setf your-name (get-string-from-user
          "Please type in your name."))

(format t "Hello, ~A!" your-name)
))
```

5. **Choose OK in the dialog item action window.**
6. **Choose Use Dialogs from the Design menu and try out your dialog box.**

Creating source code from dialogs

To preserve your dialog box, you have to save it as text in a file. The Interface Toolkit helps you by automatically generating source code from dialog boxes.

1. **To create a source file for your dialog box, choose Design Dialogs from the Design menu, then choose Print Dialog Source.**

   The source code appears in a Fred window with the default title `New`. You can edit this code and then save it to a file.

   Your file should look something like the sample code in Appendix C of this manual.

2. **Save your code to the file `hello-dialog-2.lisp`.**

Now you can close the dialog box you've created. When you open the file `hello-dialog-2.lisp` and evaluate the contents of the buffer, Macintosh Common Lisp creates a new dialog box that looks and acts like the old one. (Remember that you must choose Use Dialogs before the dialog items work.)

If you want to make further changes to that dialog box, simply choose Design Dialogs from the Design menu and continue to edit.

Remember, you must print source code before closing the dialog box if you want to save the work you've done. Closing the dialog box before printing source code discards your work.

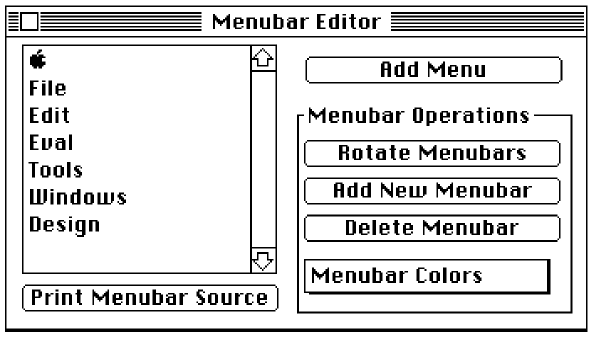
Making a menu

The Interface Toolkit also provides a Menubar Editor. With it you can make your own menus or add new menus to the usual Macintosh Common Lisp menu bar. In this section you'll add a new menu to the menu bar, create two new menu items, and copy some items from other menus.

1. **Choose Edit Menubar from the Design menu to open the Menubar Editor.**

   The Menubar Editor window appears (Figure 3-9). A floating window also appears containing the editor commands Cut, Copy, Paste, and Clear. You can use this floating window when your menubar has no Edit menu.

n **Figure 3-9** The Menubar Editor window



2. **Add a menu to the menu bar.**

   Click Add Menu at the top right. A new untitled menu appears in the list of menus and on the menu bar at the top of your screen.

3. **Give the new menu a title.**

   Click Untitled in the menu list and type `My Menu` over the highlighted text, then press Return.

4. **Add items to the menu.**

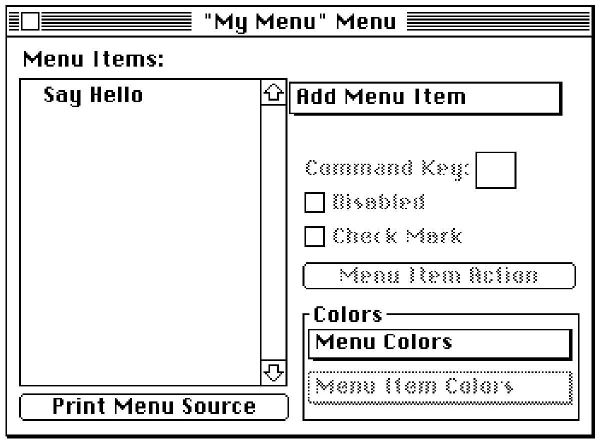
   Double-click My Menu to open an editor window.

   Click Add Menu Item to add untitled menu items to the window.

5. **Give the first menu item a name.**

   Highlight the first menu item and give it the name Say Hello (Figure 3-10). Then press Return.

n **Figure 3-10** The Menu Item Editor window



6. **Associate the first menu item with an action.**

   Open the file `hello-dialog-2.lisp` (the dialog box you just created) and select its entire contents. Press Option-W to copy them to the kill ring.

   Select the menu item, then click the Menu Item Action button. You see the familiar action item window. Highlight the line

   ```
   ;Enter action source code here.
   ```

   and press Control-Y to replace the line with the text of `hello-dialog-2.lisp`.

   Choose OK in the menu item action window.

7. **Continue adding and editing menu items.**

   The sample code in Appendix C contains another menu item that determines the factorial of an integer supplied by the user.

8. **When you are done, print the menu source code and save it to a file.**

   When you click Print Menu Source, Macintosh Common Lisp displays a window containing the source file of the menu. Edit this source file, then save it. You should have a file that resembles the sample code in Appendix C.

Creating a custom menu bar

You can also create your own menu bars for specialized development environments and applications. To make your own menu bar you can copy and move menu items from other menu bars.

1. **Choose Edit Menubar from the Design menu.**
2. **Create a new menu bar.**

   Click Add New Menubar in the Menubar Editor window. The only item on the new menu bar is the Apple menu.
3. **Add menus to the new menu bar.**

   Click Rotate Menubars to bring back the default menu bar.

   Select the File menu and copy it.

   Click Rotate Menubars again.

   Paste the File menu into the new menu bar.

   In the same way, copy the Design menu and your own menu from the default menu bar.
4. **Add and delete menu items.**

   Open the Eval menu on the default menu bar and copy the Compile File menu item from it.

   Click Rotate Menubar to access your new menu bar. Open the File menu on the new menu bar and paste Compile File into it, using Paste on the Menubar Editor floating window.

   In the File menu on the new menu bar, cut the Page Setup and Print menu items.
5. **Move menu items.**

   The menu items you added were pasted at the bottom of the menu. To change their position in the menu, you cut and paste them.

   Move Quit to the bottom of the menu by cutting and pasting it.
6. **Save your new menu bar.**

   Click Print Menubar Source, then save the file.

   You may have to edit the source code; see the next section.

Editing menu item source code

The Menu Editor is able to print source code for a menu item only if it has access to the source code of the action function of the menu item. If it doesn't, it puts `"Can't find definition"` in the place of the action function source code. You can then edit the code, putting in the real action function definition.

The source code for the action functions of some of the built-in menu items is not available. For example, if you print the source code for the File menu, you need to edit the definition of the New menu item. The definition should make an instance of whatever kind of window you want New to use; for example, if New opens a Fred window, as it does in Macintosh Common Lisp, the definition you add is `(make-instance 'fred-window)`.

If you are customizing your Macintosh Common Lisp menu bar, you may also need to edit the following definitions:

Load File	`(load (choose-file-dialog))`
Compile File	`(compile-file (choose-file-dialog :button-string "Compile"))`
Break	`(break)`
Restarts	`(ccl::choose-restarts)`
Edit Menubar	`(interface-tools::edit-menubar)`

Editing a custom menu

You can use the Menubar Editor to edit menus you've created:

1. **Open the file containing the source code of your custom menu.**

   This is the file you generated with Print Menubar Source and saved.

   Make sure your menu has a variable associated with it. To do so, embed it in the function `setf` as follows:

```
? (setf my-menu (make-instance 'menu
      ...the rest of the menu definition...
   ))
```

   Make sure you remember the additional closing parenthesis at the end of the menu definition.

2. **Evaluate the source code.**

3. **Create a new menu bar.**

4. **Install the menu in the new menu bar.**

   Type the following in the Listener:

```
? (menu-install my-menu)
```

The menu is loaded in the empty menu bar. Because the Menubar Editor does not automatically update itself, rotate the menu bars to see the new menu loaded.

For more information on menus and menu items, see Chapter 4, “Menus,” of the *Macintosh Common Lisp Reference.*

Extending the Interface Toolkit

Full source code is provided for the Interface Toolkit. You can use it as a template or extend the application for your own use.

For example, you should now be able to add a button to the Interface Toolkit that automates editing a menu source file. These are the steps to take:

- Create a button, “Edit Menu Source File.”
- Resize and rearrange the Menubar Editor dialog box so that the button fits.
- From the button, call a dialog box that prompts you for a file.
- Set a dialog item action that creates a new menu bar, loads the source code, and installs the menu on the menu bar.

What you’ve learned about prototyping

In this chapter, you’ve learned how to compile a file. You’ve prototyped a simple dialog box and added it to a menu on your own menu bar. You’ve created a custom menu bar and saved it.

You’ve learned that Lisp lends itself to an easy, incremental style of programming. You can start with simple expressions and integrate them into larger ones. You can change the environment to simplify application building. You can build tools from which to build tools.

In the next chapter, you will learn about debugging in Macintosh Common Lisp.

Chapter 4 **Debugging**

Contents

This chapter covers some of the many debugging features in Macintosh Common Lisp. If you are unfamiliar with Macintosh Common Lisp or want to refresh your memory, you should read this chapter.

Further information on debugging can be found in Chapter 10 of the *Macintosh Common Lisp Reference.*

MCL’s multiple debugging facilities

Macintosh Common Lisp provides many ways for you to examine and debug functions, source code, and environments:

- Compiler options. Macintosh Common Lisp can record the source file of all function definitions, which you can later retrieve.
- Debugging functions. The function `apropos`, also available as an item on the Tools menu, finds all symbols known to Macintosh Common Lisp whose print names contain a given string. The function `room` tells you how much space is available in the Lisp operating system, and `inspect` invokes the editable Inspector. A set of Lisp functions gives information about methods and classes.
- Fred commands. Several Fred commands help you get information about functions.
- A set of functions for signaling errors and canceling operations.
- A break-loop facility, which interrupts a program and allows you to look at the stack and examine dynamic values before returning.
- A stack backtrace.
- A single-expression stepper.
- A trace function.
- An editable Inspector.

Compiler options

You can choose whether or not to allow examination of the source file of your code. If the special variable `*record-source-file*` has the value `t`, the definition contains a pointer to the source file on disk. You can later retrieve the source-code file by pressing Option-period when the insertion point is in the symbol naming the definition, by using the Inspector, or by choosing Edit Definition from the Tools menu and typing the name.

The initial value of `*record-source-file*` is `t`.

You can look at or change this value (and many others) by choosing Environment from the Tools menu.

Debugging functions

Several Lisp functions give you useful information about interned symbols, the Lisp operating system, and classes and methods. Here are a list of those functions and an example of how each is used.

In each of these examples, assume a function `fact` with one argument, `number`, and a documentation string, `"Returns the factorial of a number."`

`(apropos` ***symbol-name-or-string*****`)`**

This Common Lisp function returns a list of all symbols known to Macintosh Common Lisp that match *symbol-name-or-string*. It corresponds to Apropos from the Tools menu.

Here is an example of the use of `apropos`, in which it is used to find every symbol that contains the string `FACT`. Notice that it brings up some constants as well as the function you're looking for.

```
? (apropos "fac")
FACT, Def: FUNCTION
CCL::REQUIRE-INTERFACE, Def: MACRO FUNCTION
CCL::PROVIDE-INTERFACE, Def: FUNCTION
REINDEX-INTERFACES, Def: FUNCTION
CCL::PROVIDE-INTERFACE, Def: FUNCTION
CCL::%REQUIRE-INTERFACE, Def: FUNCTION
CCL::FIND-INTERFACE-ENTRY, Def: FUNCTION
CCL::%FACOSH, Def: FUNCTION
CCL::LOAD-INTERFACE-FILES, Def: FUNCTION
CCL::*INTERFACES*, Value: ("TRAPS")
CCL::INTERFACE-DEFINITION-P, Def: FUNCTION
CCL::REQUIRE-INTERFACE, Def: MACRO FUNCTION
```

`(edit-definition` ***symbol-name*****`)`**

This MCL function attempts to bring up the source code of the symbol corresponding to *symbol-name*. It corresponds to Edit Definition on the Tools menu or to Meta-period.

Here is an example of the use of `edit-definition`:

```
? (edit-definition 'fact)
```

Evaluating this function will bring up the source code of `fact`.

You can retrieve the source code of the symbol only if the value of `*record-source-file*` was `t` when its file was compiled.

`(documentation` ***symbol-name type*****`)`**

This Common Lisp function displays the documentation string for the symbol corresponding to *symbol-name* (if a documentation string is available). It takes *type,* a type of symbol, as an optional second argument. It corresponds to Documentation on the Tools menu. Here is an example of the use of `documentation`:

```
? (documentation 'fact 'function)
"Returns the factorial of a number."
```

You can retrieve the documentation string of the symbol only if the value of `*save-doc-strings*` was `t` when its file was compiled.

`(arglist` ***symbol-name*****`)`**

This MCL function returns the argument list of the symbol corresponding to *symbol-name*.

Here is an example of the use of `arglist`:

```
? (arglist 'fact)
(NUMBER)
:DEFINITION
```

You can retrieve the argument list of a symbol only if the value of `*save-local-symbols*` or `*save-definitions*` was true when its definition was evaluated, or if its definition was loaded from a `fasl` file that was compiled when the value of `*fasl-save-local-symbols*` or `*fasl-save-definitions*` was true.

`(room)` This Common Lisp function reports the total size and usage of the Macintosh heap, Lisp heap, and stack. With an optional argument `(room t)`, it also reports how much of the space used contains markable objects and how much contains immediate objects.

`(inspect` ***object*****`)`**

This Common Lisp function inspects the object corresponding to *object*. It corresponds to Inspect on the Tools menu.

Here is an example of the use of `inspect`:

```
? (inspect 'fact)
```

You can even do the following:

```
? (inspect 'inspect)
```

Editing definitions

As part of debugging your code, or simply to understand what the code is doing, you often want to look at source code.

Because you'll frequently edit definitions, Macintosh Common Lisp provides a number of ways to do it. To view or edit a definition, do one of the following:

- In the Listener, type `(edit-definition '`*symbol-name*`)`, as in the previous section.
- Choose Edit Definition from the Tools menu and type the name of the definition.

  With the insertion point within or at the end of the name of the function, press Meta-period.
- Type the name of the function in the Listener, then press Meta-period.

Online documentation for supplied functions

Online documentation is available for any Common Lisp symbol and all exported Macintosh Common Lisp symbols. To see this documentation, choose Documentation from the Tools menu.

To add documentation strings to the functions and variables you write, include the documentation as a string in the second line of the definition. Here is an example of a documentation string:

```
? (defun fact (number)
  "Returns the factorial of a number."
    (if (eq 0 number) 1
        (* number (fact (- number 1)))))
FACT
```

Fred commands

A number of the preceding functions are bound to keys in Fred or to Fred commands. You can get the source code of the function (if it's available), its argument list, and its documentation string. Other Fred commands macroexpand or read the current function and provide information about the active Fred window.

For Fred commands relating to debugging, see "Help Commands" and "Lisp Operations" in Appendix A.

Error messages and functions related to errors

When Macintosh Common Lisp signals an error, it automatically provides a detailed explanation of the error and gives at least one way to recover from it. The error message tells you what kind of error occurred, in what function it occurred, what arguments the function was being applied to, and what your recovery options are.

To recover from an error:

- You can always cancel the evaluation.
- You can sometimes fix the problem by setting a value in the break loop, then continue evaluating.

You may have other options, given in the dialog box brought up by the Restarts command on the Eval menu.

Appendix B of this manual lists some common errors and what to do about them.

Reading an error message

Here is an example of an error message. The following code for creating and closing a window contains an elementary error:

```
? (setf w (make-instance 'window))
#<WINDOW "Untitled" #x4CF1D1>
? (close-window w)
> Error: Undefined function CLOSE-WINDOW called with arguments
(#<WINDOW "Untitled" #x4CF1D1>) .
> While executing: TOPLEVEL-EVAL
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry applying CLOSE-WINDOW to (#<WINDOW "Untitled"
#x4CF1D1>).
See the Restarts… menu item for further choices.
1 >
```

In this case the name of the function `window-close` is wrong and Macintosh Common Lisp doesn't recognize it. The error message first gives the nature of the problem.

```
> Error: Undefined function CLOSE-WINDOW called with arguments
(#<WINDOW "Untitled" #x4CF1D1>).
```

The message notes where the error occurred:

```
> While executing: TOPLEVEL-EVAL
```

Next Macintosh Common Lisp tells you how to recover from the error. In this case, you can either retry the evaluation with Continue, cancel it with Abort, or try Restarts for further ideas. (All three of these commands are on the Eval menu.) The error message describes what the Continue command will do.

```
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry applying CLOSE-WINDOW to (#<WINDOW "Untitled"
#x4CF1D1>).
See the Restarts… menu item for further choices.
```

Finally the error message gives you a distinctive prompt notifying you that you are in a break loop. The `1` indicates that you are in the first level of a break loop.

```
1 >
```

Recovering from an error

When an error occurs, you can always cancel the evaluation and often you can continue. The Restarts menu always allows you to cancel, and both the Continue and Abort commands also have keyboard equivalents.

The Restarts menu often provides other alternatives for recovery from an error.

Recovering from an error with Abort

When the problem is obvious, it is usually easiest to cancel out of a evaluation, using Abort, and try again.

1. **Cancel out of the evaluation.**

   Press Command-period to give the Abort command or choose Abort from the Tools menu.

2. **Edit the form.**

   Press Option-G to bring back the form as you originally typed it. Edit the form to remove the error.

3. **Retry the evaluation.**

   Press Return to try again.

Recovering from an error with Continue

When you are in the middle of a long process, such as compiling a group of 3000 files, and an error occurs late in the process, you would much rather continue than cancel. If you know that changing some value in the Macintosh Common Lisp environment will make your code run correctly, you can change values in the break loop, then continue with the evaluation.

The simple error above is properly handled by canceling and editing, but you can continue from it by defining the undefined function:

```
1 > (defun close-window (w) (window-close w))
CLOSE-WINDOW
```

If you now continue from the error, Macintosh Common Lisp closes the window and exits the break loop.

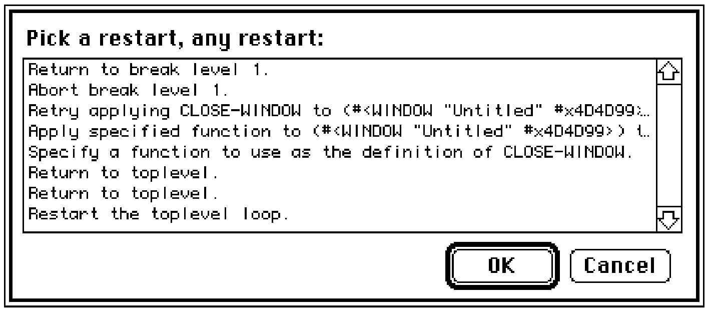
The break loop is discussed in the section “The Break Loop” later in this chapter.

Recovering from an error with Restarts

The Restarts dialog box (Figure 4-1) automatically gathers all possible ways of recovering from an error. It always offers you the option of canceling and sometimes offers the option of continuing. If there are multiple nested break loops, Restarts gives you the option of returning to any of them.

In some cases, such as the one in Figure 4-1, Restarts offers more specific suggestions. Here the option `"Apply specified function to (#<WINDOW "Untitled" #x4D4D99>)..."` lets you specify another function name.

n **Figure 4-1** A Restarts dialog box



Pick a restart, any restart:

```
Return to break level 1.
Abort break level 1.
Retry applying CLOSE-WINDOW to (#<WINDOW "Untitled" #x4D4D99>…
Apply specified function to (#<WINDOW "Untitled" #x4D4D99>) t…
Specify a function to use as the definition of CLOSE-WINDOW.
Return to toplevel.
Return to toplevel.
Restart the toplevel loop.
```

The following shows you how to use the Restarts dialog box.

1. **Cause an error.**

   Type the code fragment that signals the error. First make a window:

   ```
   ? (setf w (make-instance 'window))
   ```

   Then make a mistake closing it:

   ```
   ? (close-window w)
   ```

   You should get an error message. (If you've already redefined `close-window`, use another symbol, such as `(get-rid-of-window w)`.)

2. **Now find the correct function name.**

   (Assume you don't know it.)

   Select Apropos from the Tools menu or use the function `apropos`. Search on a likely symbol, such as `close`. Evaluating `(apropos "close")` produces many symbols, among them:

```
INSPECTOR::DELAYED-WINDOW-CLOSE, Def: STANDARD-GENERIC-FUNCTION
INSPECTOR::CLOSURE-N-CLOSED, Def: STANDARD-GENERIC-FUNCTION...
WINDOW-CLOSE, Def: STANDARD-GENERIC-FUNCTION
_CLOSEPICTURE, Def: MACRO FUNCTION,  Value: 43252
STREAM-CLOSE, Def: STANDARD-GENERIC-FUNCTION
WINDOW-CLOSE-EVENT-HANDLER, Def: STANDARD-GENERIC-FUNCTION
_CLOSERESFILE, Def: MACRO FUNCTION,  Value: 43418
_PRCLOSE, Def: MACRO FUNCTION,  Value: 3489660928
ED-MOVE-OVER-CLOSE-AND-REINDENT, Def: STANDARD-GENERIC-FUNCTION
```

   Examining this list, you find a promising candidate, the function `window-close`. (As you can see, there may be other possible candidates. Use the online documentation to help select the one you want. Don't be afraid to play around.)

3. **Choose the Restarts menu item from the Eval menu and select `"Apply specified function..."`.**

4. **Type the name of the function, `window-close`.**

   Macintosh Common Lisp exits the break loop and closes the window.

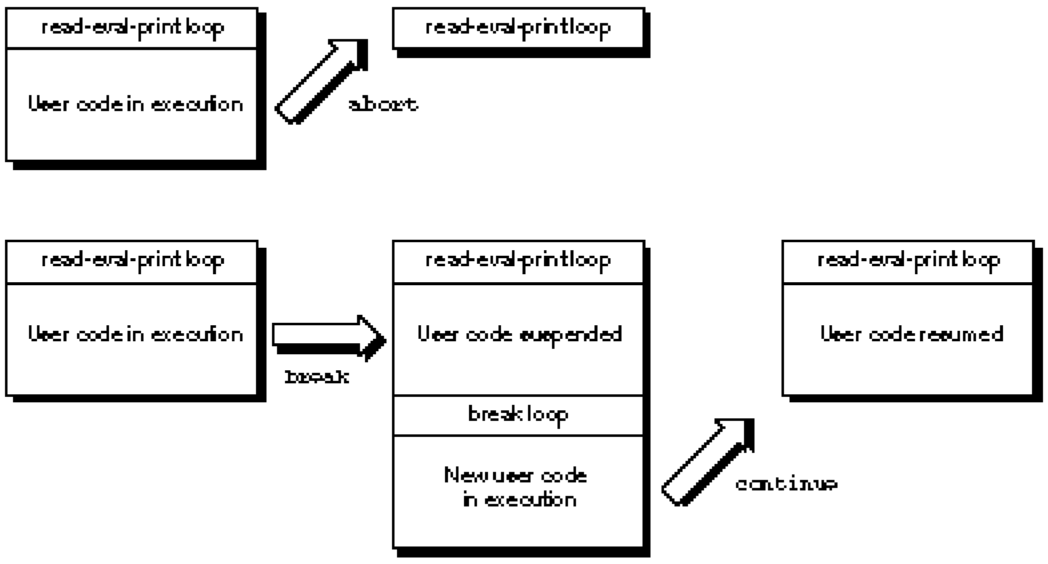
Since the Restarts menu is generated by a fixed algorithm, it often suggests the same restart more than once. It doesn't matter which one you choose.

The break loop

The break loop is a read-eval-print loop that comes into play when an error occurs or when you explicitly call `(break)`. A break loop acts like the normal read-eval-print loop, except that it suspends your program, allowing you to interact with Macintosh Common Lisp on top of your program. From a break loop you can resume the program or cancel it.

Break loops themselves can have break loops. That is, from the normal read-eval-print loop you can break to a break loop, from that loop to another break loop, and so on. Each level of break loop adds a new area to the stack; canceling or continuing out of a break loop removes its stack area (Figure 4-2).

n **Figure 4-2** Effects on the stack of Break, Abort, and Continue



Break loops add new areas to the stack, but Abort and Continue remove areas from the stack. New areas are added to the bottom. Within a break loop, the normal question mark prompt is replaced by a number and an angle bracket. The number of the prompt represents the level of the break loop.

Because the break loop runs on top of the interrupted program, all globally defined and special variables have the values they had when the interrupted program was suspended. Within the break loop you can redefine functions, write methods, and change the values of global and special variables (but not variables defined only for the duration of a function, because they have values only within the program that has been suspended).

By changing values within the break loop, you can often continue from an error.

Here is a simple example of using a break loop. The special variable `*backtrace-on-break*` determines whether the Stack Backtrace window opens automatically within a break loop. You can define a function to enter the break loop if the value of `*backtrace-on-break*` is `t`. Otherwise, the function prints the phrase "The Stack Backtrace window does not open automatically."

1. **Define a function that may enter a break loop.**

   Remember, you type only what is in bold. Press Return to get the indicated response from Macintosh Common Lisp.

   ```
   ? (defun take-a-break ()
      (if (eq *backtrace-on-break* t)
       (break)
       (format t "The Stack Backtrace window does not open
            automatically.")))
   TAKE-A-BREAK
   ```

2. **Set it up to enter a break loop.**

   ```
   ? (setf *backtrace-on-break* t)
   T
   ```

   Since the value of `*backtrace-on-break*` is `t`, calling `take-a-break` causes Macintosh Common Lisp to enter a break loop and display a Stack Backtrace window.

3. **Call the function.**

   ```
   ? (take-a-break)
   >Break:
   > While executing: TAKE-A-BREAK
   > Type Command-/ to continue, Command-. to abort.
   > If continued: Return from BREAK.
   See the Restarts… menu item for further choices.
   ```

4. **Change the value of `*backtrace-on-break*` inside the break loop.**

   ```
   1 > (setf *backtrace-on-break* nil)
   NIL
   ```

5. **Continue from the break.**

   Even after you fix the problem that is causing the break, Macintosh Common Lisp remains in the break loop until you choose Abort, Continue, or a selection from the Restarts dialog box that performs a cancellation or continuation.

   ```
   1 > (continue)
   NIL
   ```

6. **Now try the function again.**

   ```
   ? (take-a-break)
   The Stack Backtrace window does not open automatically.
   ```

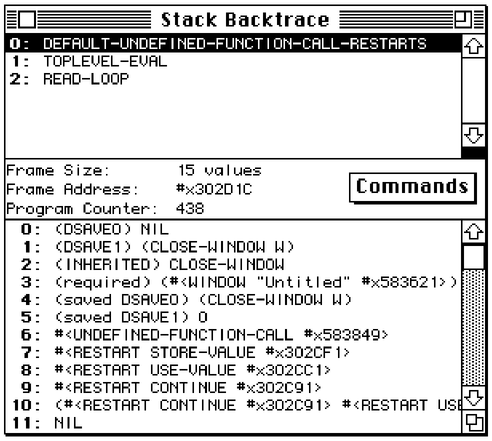
You can get into the break loop at any time by selecting Break from the Eval menu or by pressing Command-comma.

The stack backtrace

Whenever Macintosh Common Lisp is in a break loop, the stack backtrace is enabled, letting you examine every value on the stack. If the value of `*backtrace-on-break*` is `t`, the Stack Backtrace window opens automatically when Macintosh Common Lisp enters a break loop. If the value of `*backtrace-on-break*` is `nil`, you can bring up the Stack Backtrace window by choosing Backtrace from the Tools menu.

Stack backtracing is the second line of defense after reading the error messages. It lists all the functions pending on the stack at the time of the error (not necessarily all the functions that have been called—use `trace` for that). By examining what is on the stack and comparing it to what you expect, you can often determine how the error occurred. By inspecting the function containing the error, you may be able to edit and correct the problem.

n **Figure 4-3** A stack backtrace



```
0:  DEFAULT-UNDEFINED-FUNCTION-CALL-RESTARTS
1:  TOPLEVEL-EVAL
2:  READ-LOOP
```

```
Frame Size:        15 values
Frame Address:     #x302D1C
Program Counter:   438
```

```
 0:  (DSAVE0) NIL
 1:  (DSAVE1) (CLOSE-WINDOW W)
 2:  (INHERITED) CLOSE-WINDOW
 3:  (required) (#<WINDOW "Untitled" #x583621>)
 4:  (saved DSAVE0) (CLOSE-WINDOW W)
 5:  (saved DSAVE1) 0
 6:  #<UNDEFINED-FUNCTION-CALL #x583849>
 7:  #<RESTART STORE-VALUE #x302CF1>
 8:  #<RESTART USE-VALUE #x302CC1>
 9:  #<RESTART CONTINUE #x302C91>
10:  (#<RESTART CONTINUE #x302C91> #<RESTART USE
11:  NIL
```

There are two tables in the Stack Backtrace window (Figure 4-3). The upper table shows you the functions pending on the stack. By clicking any function once, you can examine its stack frame; by clicking it twice, you can inspect it.

The lower table shows the stack frame of the function that is selected in the upper table.

In the space between the tables are three pieces of information about the frame: the number of values in the frame, the memory address of the frame, and the program counter within the function where execution has been suspended. These are useful for low-level system debugging and for reporting any bugs to Apple Computer, Inc.

The stepper

Macintosh Common Lisp provides the `step` macro as a simple way of stepping through a single form, expression by expression.

You can use the `step` macro on compiled functions only if their definitions have been retained. Function definitions are retained if the function is compiled with the variable `*save-definitions*` set to `t`. If the function was compiled with `*save-definitions*` set to `nil`, the value must be changed and the function must be recompiled before it can be stepped through.

The `step` macro is usually called only from the top level. You can invoke internal stepping through `trace`.

It is not generally possible to evaluate code that requires the use of `without-interrupts` or code that uses the Macintosh graphics interface.

To see how the stepper works, step through `take-a-break`:

1. **Make sure that `take-a-break` has been compiled with `*save-definitions*` set to `t` .**

   You can check this by seeing whether `*save-definitions*` is selected in the Environment dialog box on the Tools menu.

   If it was not selected, select it.

   In the same Environment dialog box, make sure that `*backtrace-on-break*` is selected.

   Then evaluate `take-a-break` again.

2. **Step through `take-a-break`.**

```
? (step (take-a-break))
```

You can also try stepping through other functions, such as `fact`.

Trace

Tracing is useful when you want to find out why a function behaves in an unexpected manner, perhaps because incorrect arguments are being passed.

Tracing causes actions to be taken when a function is called and when it returns. The default tracing actions print the function name and arguments when the function is called and print the values returned when the function returns.

Other actions can be specified. These include entering a break loop when the function is entered or exited, or stepping the function.

```
? (defun fact (num)
    (declare (notinline fact)) ; this is important for tracing
fact
     (if (= num 0)
       1
       (* num (fact (- num 1)))))
FACT
```

Here the trace macro is used on `fact`:

```
? (trace fact)
NIL
? (fact 5)
 Calling (FACT 5)
  Calling (FACT 4)
   Calling (FACT 3)
    Calling (FACT 2)
     Calling (FACT 1)
      Calling (FACT 0)
      FACT returned 1
     FACT returned 1
    FACT returned 2
   FACT returned 6
  FACT returned 24
 FACT returned 120

120
```

To turn `trace` off, use `untrace` on the same function:

```
? (untrace fact)
(FACT)
```

The Inspector

The Inspector lets you look quickly at any component of one or more data objects. It provides special inspection facilities for the current state of the system data. The Inspector is readily available from most other help and debugging windows such as Apropos.

Double-clicking on an object displayed in an Inspector window brings up another Inspector window displaying the components of that object.

Because objects are editable in Inspector windows, you can change the state of system internals and other components on the fly. However, you should modify an object with the Inspector only when you understand how it works and what effect your modification will have. Otherwise you may corrupt your environment or even cause the system to crash.

For example, it is safe to set the value of a global variable in the Inspector window when inspecting a symbol. However, it's inadvisable to use the Inspector to set the values of slots in objects; use the standard interface functions instead.

To see the Inspector, choose Inspect from the Tools menu. You can also call `inspect` on a Lisp object or use the Fred command Control-X Control-I. When you choose Apropos from the Tools menu and double-click a symbol name, Macintosh Common Lisp creates an Inspector window containing information about that symbol.

The following sections show some examples of using the Inspector.

Inspecting a data object with `inspect`

To inspect a data object, call the function `inspect` on it. Here is an example of inspecting a window `w` that has already been closed.

1. **First create a window.**

```
? (setq w (make-instance 'window))
#<WINDOW "Untitled" #x583621>
```

2. **Then close it.**

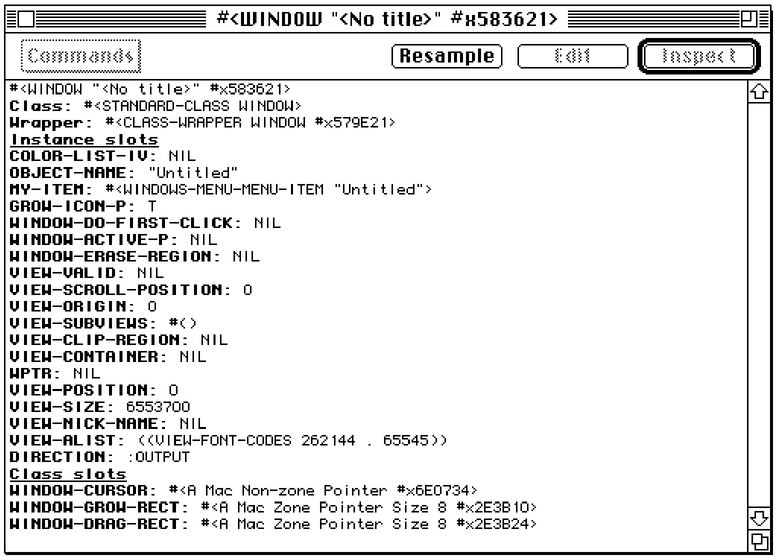
```
? (window-close w)
NIL
```

3. **Inspect it.**

```
? (inspect w)
#<INSPECTOR-WINDOW "#<WINDOW \"<No title>\" #x583621>" #x452511>
```

An Inspector window opens to inspect `w` (Figure 4-4). The `nil` value in its `wptr` slot shows that the window `w` is closed.

n **Figure 4-4** An Inspector window showing components of a window



```
#<WINDOW "<No title>" #x583621>
Class: #<STANDARD-CLASS WINDOW>
Wrapper: #<CLASS-WRAPPER WINDOW #x579E21>
Instance slots
COLOR-LIST-IV: NIL
OBJECT-NAME: "Untitled"
MY-ITEM: #<WINDOWS-MENU-MENU-ITEM "Untitled">
GROW-ICON-P: T
WINDOW-DO-FIRST-CLICK: NIL
WINDOW-ACTIVE-P: NIL
WINDOW-ERASE-REGION: NIL
VIEW-VALID: NIL
VIEW-SCROLL-POSITION: 0
VIEW-ORIGIN: 0
VIEW-SUBVIEWS: #()
VIEW-CLIP-REGION: NIL
VIEW-CONTAINER: NIL
WPTR: NIL
VIEW-POSITION: 0
VIEW-SIZE: 6553700
VIEW-NICK-NAME: NIL
VIEW-ALIST: ((VIEW-FONT-CODES 262144 . 65545))
DIRECTION: :OUTPUT
Class slots
WINDOW-CURSOR: #<A Mac Non-zone Pointer #x6E0734>
WINDOW-GROW-RECT: #<A Mac Zone Pointer Size 8 #x2E3B10>
WINDOW-DRAG-RECT: #<A Mac Zone Pointer Size 8 #x2E3B24>
```

You can inspect this window and its components, for instance #<STANDARD-CLASS WINDOW>. Again, however, don't try to edit its values here (although Macintosh Common Lisp won't stop you). For example, never try to reopen a window by giving the wptr slot some value; such edits will crash your system and may actually damage it. Even milder edits, like changing the colors of parts of a window, will not appear correctly in the window because the MCL interface function that changes those colors, set-part-color, does more than change the value of a slot.

Use the editing ability of the Inspector to change global variables and make other simple changes; to change values in complex objects, use the interface functions documented in the *Macintosh Common Lisp Reference*.

Inspecting objects from debugging windows

In Macintosh Common Lisp, you can inspect objects from many windows associated with debugging. For example, from the Apropos window you can double-click a symbol to inspect it. You can double-click any of the symbol's components to inspect that component.

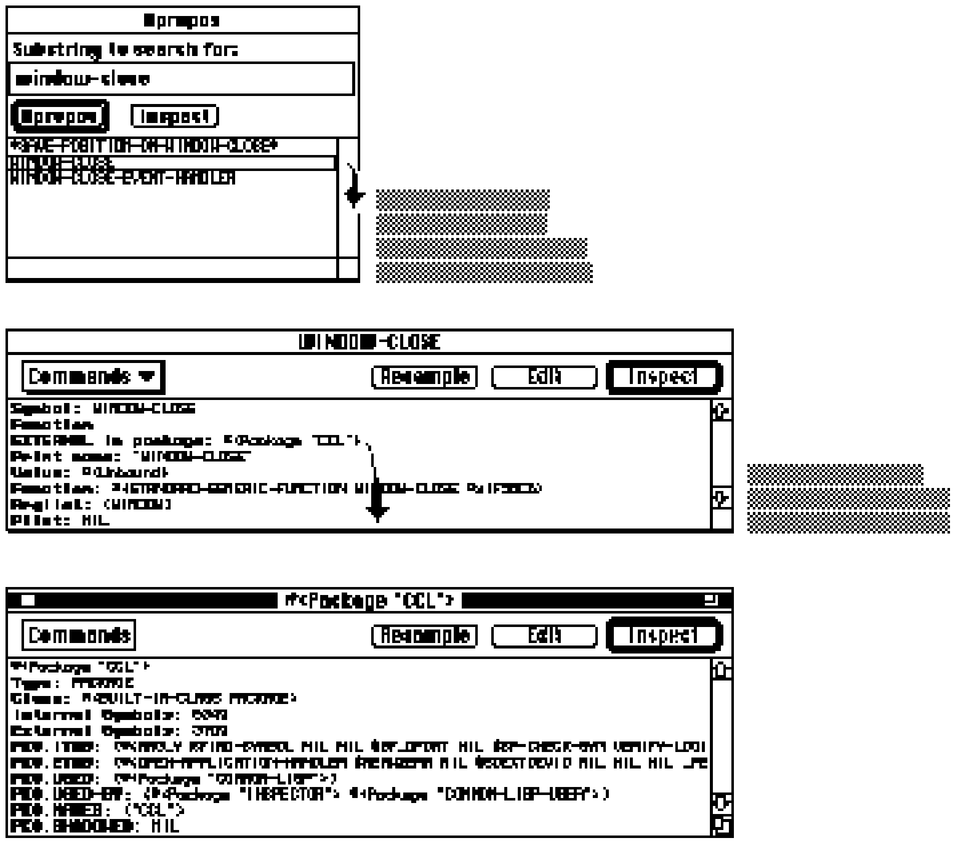
Figure 4-5 shows a typical use of the Inspector with the Apropos window. The user first searches in the Apropos window for window-close; Apropos shows the symbol window-close and several other symbols.

Double-clicking window-close opens a window titled WINDOW-CLOSE. This window shows the print name and package of the symbol window-close. You can inspect any of the objects in the window by double-clicking that object.

In Figure 4-5, the user double-clicks the package #<PACKAGE "CCL"> to inspect that package.

From the Stack Backtrace window, you can double-click any object to inspect it. You can also inspect any symbol in an editor or Listener window by placing the insertion point within or just past it and pressing Control-X Control-I.

n **Figure 4-5** An Apropos window and the Inspector



```
*SAVE-POSITION-ON-WINDOW-CLOSE*
WINDOW-CLOSE
WINDOW-CLOSE-EVENT-HANDLER
```

```
Symbol: WINDOW-CLOSE
Function
EXTERNAL in package: #<Package "CCL">
Print name: "WINDOW-CLOSE"
Value: #<Unbound>
Function: #<STANDARD-GENERIC-FUNCTION WINDOW-CLOSE #x[illegible]>
Arglist: (WINDOW)
Plist: NIL
```

```
#<Package "CCL">
Type: PACKAGE
Class: #<BUILT-IN-CLASS PACKAGE>
Internal Symbols: [illegible]
External Symbols: [illegible]
PKG.ITAB: (#(APPLY %FIND-SYMBOL NIL NIL %DEF-DEFUN NIL %SP-CHECK-SYM VERIFY-LOO
PKG.ETAB: (#(OPEN-APPLICATION-HANDLER %REMAREA NIL %DEXTDEVID NIL NIL NIL _FE
PKG.USED: (#<Package "COMMON-LISP">)
PKG.USED-BY: (#<Package "INSPECTOR"> #<Package "COMMON-LISP-USER">)
PKG.NAMES: ("CCL")
PKG.SHADOWED: NIL
```

What you’ve learned about debugging

Macintosh Common Lisp has unusually robust and well-integrated debugging tools, including detailed error messages, automatically generated recovery procedures, and code-inspection features that let you edit values on the fly. Debugging commands such as Apropos, Inspect, and Backtrace are available from the Tools menu.

The next chapter tells you where to go for further information

- if you’re learning Lisp
- if you’re learning CLOS, the Common Lisp Object System
- if you’re learning about programming the Macintosh computer
- if you’re updating from a previous version of Macintosh Common Lisp
- if you need examples of CLOS code
- if you’d like to communicate with other Macintosh Common Lisp users and developers
- if you’re doing commercial or educational development with Macintosh Common Lisp

Chapter 5 **Where to Go From Here**

Contents

The following sections describe useful background reading and sources of information.

The single most valuable source

At least until the release of the draft ANSI specification, the second edition of *Common Lisp: The Language* is the standard reference manual and language specification for CLOS, as well as the most up-to-date specification for all of Common Lisp. It is not a tutorial but is a must for every Lisp user.

Steele, Guy L., and others. *Common Lisp: The Language,* second edition. Maynard, MA: Digital Press, 1990.

If you are learning Lisp

Many good tutorials exist on Common Lisp. Here is a selection of the best:

Brooks, Rodney. *Programming in Common Lisp.* New York: John Wiley & Sons, 1985.

Koschmann, Timothy. *The Common Lisp Companion*. New York: John Wiley & Sons, 1990.

This well-written book includes material from the second edition of *Common Lisp: The Language,* including information on CLOS. The author is on the ANSI XJ313 standards committee for Common Lisp.

Miller, Molly M., and Eric Benson. *Lisp Style and Design.* Maynard, MA: Digital Press, 1990.

An excellent intermediate-level book for someone learning Common Lisp.

Norvig, Peter. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* San Mateo, CA: Morgan-Kauffmann, 1991.

An excellent book for all levels of Common Lisp programming, with many examples.

Tatar, Deborah. *A Programmer's Guide to Common Lisp*. Maynard, MA: Digital Press, 1987.

Touretzky, David. *Common Lisp: A Gentle Introduction to Symbolic Computing.* Reading, MA: Addison-Wesley, 1990.

Touretzky writes well and includes many good examples for novices.

Wilensky, Robert. *Common LispCraft*. New York: Norton & Co, 1986.

Winston, Patrick, and Berthold Claus Paul Horn. *Lisp,* third edition. New York: Harper & Row, 1989.

A thorough academic text that includes material on CLOS.

The following is possibly the best book ever written on computer programming. It uses Scheme, a language similar to Lisp. With a relatively small knowledge of Lisp, you should be able to make use of it.

Abelson, Harold, and Gerald Sussman. *Structure and Interpretation of Computer Programs.* Cambridge, MA: MIT Press, 1985.

If you are learning CLOS

The following book by Sonya Keene is generally reckoned to be the most thorough CLOS tutorial on the market. Keene was also a contributor to the second edition of *Common Lisp: The Language.*

Keene, Sonya E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Reading, MA: Addison-Wesley, 1989.

The books by Koschmann and by Winston and Horn, listed in the preceding section, also include material on CLOS.

If you’re learning about Macintosh programming

If you want to know more about programming the Macintosh, Apple Developer University offers excellent self-paced courses for novice and intermediate Macintosh programmers, including courses on Macintosh Programming Fundamentals and Introduction to Object-Oriented Programming. Other “live” courses are also offered through Developer University.

The standard reference for Macintosh programming is *Inside Macintosh*, published by Apple Computer, Inc., and available through APDA (see the later section on APDA). *Inside Macintosh* is available on paper, online, and on several CD-ROM (compact disc read-only memory) discs.

If you are programming high-level Lisp code for porting, you may not need to use *Inside Macintosh* as a reference. However, *Inside Macintosh* is indispensable for programming all low-level Macintosh features.

If you’re updating from a previous version of Macintosh Common Lisp

If you’re not familiar with the Common Lisp Object System, read Appendix C, “The Common Lisp Object System,” in the *Macintosh Common Lisp Reference*. Appendix D, “Converting Your Files to CLOS,” of the *Macintosh Common Lisp Reference* is a guide to converting your code to the new object standard. (The experience of early MCL users is that converting code is not nearly so daunting as thinking about it.)

The Common Lisp language is documented in *Common Lisp: The Language*. The Macintosh Common Lisp programming environment, class library, and extensions to Common Lisp are documented in the *Macintosh Common Lisp Reference*.

If you would like examples of CLOS code

The Examples folder of Macintosh Common Lisp provides examples of CLOS programming. Continuing discussions of CLOS programming issues take place on the `info-mcl` mailing list. See the next section, "If You'd Like to Communicate With Other Macintosh Common Lisp Programmers."

This mailing list, `info-mcl`, is now also available as a Usenet news group. Information moves transparently between the Internet and Usenet groups; if sent to one, it appears on both.

If you'd like to communicate with other Macintosh Common Lisp programmers

Apple Computer, Inc., supports Macintosh Common Lisp developers through an Internet mailing list and Usenet news group, `info-mcl`. This news group posts answers to specific technical questions and provides a forum in which Macintosh Common Lisp developers can communicate. Apple also provides an `info-mcl` archive and a library of extensions, third-party code, and patches, which are available by anonymous FTP.

FTP Host: `cambridge.apple.com`

User: `anonymous`

Password: <anything non-blank>

`/pub/mcl2/contrib/` contains the contributed code

`/pub/mail-archive/` contains the `info-mcl` archive

You may send your own code for posting on `info-mcl`. (Apple reserves the right to distribute this code to other users unless specifically requested not to.)

Access to the `info-mcl` bulletin board is also available through online services, such as CompuServe, that provide a mail gateway to Internet.

The `info-mcl` archives are available on AppleLink, under "Developer Support: Developer Talk: Macintosh Developer Tool Discussion: Macintosh Common Lisp Discussion."

Communicating through Internet

To sign up for `info-mcl`, send a request to

```
info-mcl-request@cambridge.apple.com
```

Send messages to `info-mcl` at this address:

```
info-mcl@cambridge.apple.com
```

Communicating through Usenet

To sign up for Usenet, get into UNIX® `rn` and give this command:

```
g comp.lang.lisp.mcl
```

Send messages through mail to `info-mcl`, or through `rn`.

Communicating through AppleLink

You can communicate with the Internet through AppleLink. To sign up for `info-mcl`, send a request to this AppleLink address:

```
info-mcl-req@cambridge.apple.com@internet#
```

Send messages to `info-mcl` at this AppleLink address:

```
info-mcl@cambridge.apple.com@internet#
```

Communicating through CompuServe

Sign up for `info-mcl` by sending mail to

```
>INTERNET: info-mcl-request@cambridge.apple.com
```

Send messages to `info-mcl` from CompuServe at the following address:

```
>INTERNET: info-mcl@cambridge.apple.com
```

Communicating through other services

Some other communications services have links with Internet. Please check with the customer services group at your preferred communications service.

If you’re a developer–APDA Developer Services

APDA, Apple’s source for developer tools, offers worldwide access to a broad range of programming products, resources, and information for anyone developing on Apple platforms. You’ll find the most current versions of Apple and third-party development tools, debuggers, compilers, languages, and technical references for all Apple platforms. To establish an APDA account, obtain additional ordering information, or find out about site licensing and developer training programs, please contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014-6299

800-282-2732 (United States)
800-637-0029 (Canada)
408-562-3910 (International)
Fax: 1-0408-562-3971
Telex: 171-576
AppleLink address: APDA

If you provide commercial products and services, please call 408-974-4897 for information on the developer support programs available from Apple Computer, Inc.

Appendix A **Fred Commands**

Contents

In this appendix you'll learn about the Fred modifier keys and the standard Fred commands.

Like all parts of Macintosh Common Lisp, Fred is written in Macintosh Common Lisp and is fully extensible. You can change what key runs any of these commands and can write commands of your own.

For full details about these commands, see Chapter 2, "Editing in Macintosh Common Lisp," and Chapter 14, "Programming the Editor," both in the *Macintosh Common Lisp Reference.*

Fred modifier keys

Fred relies on two modifier keys to indicate command keystrokes. In Fred, these modifiers are called Control and Meta. In Macintosh Common Lisp, the following keystrokes are used to invoke Meta and Control sequences:

- The Fred Control key is the Macintosh Control key. If your Macintosh computer does not have a Control key, you can use the Command key. For details, see "The Control and Meta Modifier Keys" in Chapter 2, "Editing in Macintosh Common Lisp," of the *Macintosh Common Lisp Reference*.)

  In the following tables, whichever key you are using to indicate Control is referred to as the Control key.

  To enter a Control command keystroke, hold down the Control key while you press the other key of the keystroke. For example, to enter Control-X, hold down the Control key and press X. To enter Control-X Control-S (Save), hold down the Control key and press X, then continue to hold down Control and press S. To enter Control-X H (Select Entire Buffer), hold down Control and press X, then release the Control key and press H.

- The Fred Meta key is the Macintosh Option key. By loading the example file `escape-key.lisp`, discussed in Chapter 2 of the *Macintosh Common Lisp Reference*, you can reassign Meta to the Escape key.

  To enter a Meta command keystroke, hold down the Option key while you press the other key of the keystroke. For example, to enter Meta-X, hold down the Meta key and press X. This differs from some other implementations of Emacs, in which you press the Meta key, then the command letter. To enter a Control-Meta keystroke, hold both modifier keys down at once as you press the other key of the keystroke.

Help commands

The help functions are bound to the keystroke sequences shown in Table A-1.

n **Table A-1** Help command keystrokes

Function	**Keystroke**
Displays Fred Help window with list of all keyboard commands	Control-?
Displays definition of current expression	Meta-period
Inspects current expression	Control-X Control-I
Prints argument list of current expression	Control-X Control-A
Prints documentation string of current expression	Control-X Control-D
Prints information about current Fred window	Control-=

Movement

During editing, use the keystrokes shown in Table A-2 to move the insertion point.

n **Table A-2** Movement command keystrokes

Function	**Keystroke**
Moves backward one character	Control-B, ←
Moves forward one character	Control-F, →
Moves backward one word	Meta-B, Meta-←
Moves forward one word	Meta-F, Meta-→
Moves forward one s-expression	Control-Meta-F, Control-→
Moves backward one s-expression	Control-Meta-B, Control-←
Moves to beginning of line	Control-A
Moves to end of line	Control-E
Moves to beginning of current top-level s-expression	Control-Meta-A
Moves to end of current top-level s-expression	Control-Meta-E
Moves up one line (to previous line)	Control-P, ↑
Moves down one line (to next line)	Control-N, ↓
Moves forward one screen	Control-V
Moves backward one screen	Meta-V
Moves to beginning of buffer	Meta-<
Moves to end of buffer	Meta->
Moves over next close parenthesis and reindents	Meta-)

Selection

The keystroke sequences shown in Table A-3 are used to select text.

n **Table A-3** Selection command keystrokes

Function	**Keystroke**
Selects current expression	Control–Meta–Space bar
Selects current top-level expression (the expression that has an open parenthesis flush with the left margin)	Control-Meta-H
Selects entire buffer	Control-X H

Insertion

The sequences shown in Table A-4 are used to insert text and space.

n **Table A-4** Insertion command keystrokes

Function	**Keystroke**
Inserts new line without moving insertion point	Control-O
Reindents current line or selection	Tab
Reindents current expression	Control-Meta-Q
Inserts Return followed by Tab	Control-Return
Yanks (pastes) current kill-ring string	Control-Y
Rotates the string that is pasted from the kill ring	Meta-Y

(continued)

n **Table A-4** Insertion command keystrokes (continued)

Function	Keystroke
Quotes next keystroke, allowing access to Macintosh optional character set; use with Option key and control characters such as Tab	Control-Q
Inserts quotation marks and moves insertion point between them to type a string	Meta-"
Inserts sharp comment signs and moves insertion point between them to type a sharp comment	Meta-#
Inserts set of parentheses and moves insertion point between them to type an expression	Meta-(
Converts rest of current word or selection to uppercase	Meta-U (Uppercase U)
Converts rest of current word or selection to lowercase	Meta-L
Capitalizes rest of current word or selection	Meta-C
Transposes the two characters on either side of insertion point	Control-T
Transposes the two words on either side of insertion point	Meta-T
Transposes the two s-expressions on either side of insertion point	Control-Meta-T

Deletion

The sequences shown in Table A-5 are used to delete text and spaces.

n **Table A-5** Deletion command keystrokes

Function	**Keystroke**
Deletes character to left of insertion point	Delete
Deletes word to left of insertion point	Meta-Delete
Deletes expression to left of insertion point	Control-Meta-Delete
Deletes character to right of insertion point	Control-D, Forward Delete on Apple Extended Keyboard
Deletes word or part of word to right of insertion point, adds to kill ring	Meta-D
Deletes from insertion point to end of line, adds to kill ring	Control-K
Deletes expression to right of insertion point, adds to kill ring	Control-Meta-K
Deletes current selection, adds to kill ring	Control-W
Copies current selection onto kill ring without deleting	Meta-W
Deletes whitespace (spaces or tabs) from insertion point to next nonwhite character	Control-X Control–Space bar
Deletes whitespace (spaces or tabs) around insertion point, adds one space	Meta–Space bar

Lisp operations

The keystroke sequences in Table A-6 are used to evaluate, compile, macroexpand, and read the current Lisp expression.

n **Table A-6** Lisp operation command keystrokes

Function	**Keystroke**
Evaluates current expression	Enter
Evaluates or compiles current selection or current top-level expression	Control-X Control-C
Evaluates current expression even if `*compile-definitions*` is true	Control-X Control-E
Repeatedly macroexpands the current expression until the result is no longer a macro. Pretty-prints the result of each expansion in the Listener.	Control-M
Macroexpands the current expression and pretty-prints it in the Listener	Control-X Control-M
Reads the current expression and pretty-prints it in the Listener	Control-X Control-R

Windows

The keystroke sequences in Table A-7 are used to save and select text manipulated in windows.

n **Table A-7** Window command keystrokes

Function	**Keystroke**
Saves contents of active Fred window to file	Control-X Control-S
Saves contents of active Fred window under a new name	Control-X Control-W
Selects a text file and opens a Fred window to edit that file	Control-X Control-V

Incremental search

A search is said to be incremental when the searching process begins as soon as you begin to enter the search string. For instance, if you are searching for the word `defclass`, Macintosh Common Lisp finds the first `d` as soon as you type it. When you add `e`, it finds the first occurrence of `de`, and so on. This provides autocompletion for the search process.

Full details on searching can be found in the section “Incremental Searching in Fred” in “Editing in Macintosh Common Lisp,” Chapter 2 of the *Macintosh Common Lisp Reference.*

The keystroke sequences in Table A-8 are used during incremental searches.

n **Table A-8** Search command keystrokes

Function	**Keystroke**
Searches forward incrementally	Control-S
Searches backward incrementally	Control-R
Deletes characters from search string	Delete
Terminates incremental search	Escape
Cancels incremental search	Control-G
Inserts quoted character	Control-Q
Copies word following insertion point into search string	Control-W
Copies line following insertion point into search string	Control-Y

Appendix B Common Errors and What to Do About Them

Contents

Every programmer has a personal set of favorite errors, and the following list is not comprehensive. However, a few errors appear frequently. This appendix lists the ones that appear most often and shows how to recover from them.

If you are having trouble with an error, use the debugging techniques explained in Chapter 4 and look at the syntax of the functions you
are using.

If you still can't fix the error, talk to a fellow Macintosh Common Lisp programmer. One way of doing that is to post the error on `info-mcl` (see Chapter 5, "Where to Go From Here"). Please include code samples if possible.

Errors

The following error messages and situations are associated with common and easily fixed problems in Macintosh Common Lisp.

Nothing happens after you type an expression and press Return

Here are some possible explanations and solutions:

- You may not have completed the expression. If you have completed the expression, the parenthesis at the beginning of the expression blinks when the insertion point is just after the last parenthesis.

  If the top-level parenthesis isn't blinking, continue adding parentheses at the appropriate places until it does.
- You may be in a Fred window instead of the Listener.

  Select the expression and press Command-E to evaluate.
- An expression may simply take significant time to evaluate—for example, `(fact 10000)`.

  A large compilation takes time.

  As long as the one-line minibuffer at the bottom of the screen shows Macintosh Common Lisp is busy, your work is continuing. If you are evaluating a very large buffer or group of buffers and want to make sure that you are progressing, set the value of the variable `*verbose-eval-selection*` to `t`. This will send results of intermediate evaluations to the Listener.

File *filename* does not exist

If you get the error message "`File` *filename* `does not exist`", here are some possible explanations and solutions:

- You may have mistyped the pathname or used the wrong pathname syntax.

  Check the pathname and try again, or use the Open command on the File menu or the Load command on the Eval menu.

Unbound variable

If you get the error message “`Unbound variable` *symbol-name*”, here are some possible explanations and solutions:

- You may have mistyped the symbol.

  Check it and try again.
- You may have defined the symbol locally, within a function, rather than as a global symbol.
- The symbol may be defined in another package.

Important The package problem occurs frequently in conversions from the older Common Lisp standard. The new draft standard of Common Lisp uses the package name `common-lisp`, whereas older code uses the package name `lisp`.

No applicable primary method for args

If you get the error message “`No applicable primary method for args to` *function*”, here are a possible explanation and solution:

- The function you are using is a generic function, which has different methods for different classes of arguments. It does not have a method for the class of the argument you are using. For example, you may have applied a function to a button dialog item that has methods only for table dialog items.

  Use another function or write a method. See *Common Lisp: The Language* and Appendix C of the *Macintosh Common Lisp Reference* for further information on generic functions and methods.

Undefined function called with arguments

If you get the error message "`Undefined function` *function* `called with arguments`", here are a possible explanation and solution:

- The function you called doesn't exist. You probably have mistyped or misremembered the name.

  Use the Apropos command to look through the symbol names until you find the right one.

  Remember that some function names have changed in the new standard. If the function is a Common Lisp function, check *Common Lisp: The Language*.

Wrong number of arguments

If you get the error message "`Wrong number of arguments`", here are a possible explanation and solution:

- You have given the function too many or too few arguments.

  Use `(arglist '`*function-name*`)` to check the function's argument list and try again.

  You may want to set your environment so that the minibuffer shows you a function's argument list after you type its name. Choose Environment from the Tools menu and put a checkmark next to `*arglist-on-space*`, or evaluate the following:

  ```
   ? (setf *arglist-on-space* t)
  ```

Value is not of the expected type

If you get the error message "`value` *`value`* `is not of the expected type`", here are a possible explanation and a solution:

- The function (or a function it calls) was expecting an argument of a different data type. Changing the data type of the top-level function's arguments is not always the right solution; often the problem is in one of the called functions.

  Use the Stack Backtrace window to see where the error occurred.

Appendix C **Code Listings for Chapter 3**

Contents

The following sections show how the code you developed in Chapter 3 should look. However, the numbers indicating the exact location of your dialog items will probably vary from those given here.

Dialog box code

Here is a code listing for the dialog box in Chapter 3:

```
(DEFVAR YOUR-NAME)

(MAKE-INSTANCE 'DIALOG
  :WINDOW-TYPE :DOCUMENT :WINDOW-TITLE "Good morning to you!"
  :VIEW-POSITION '(:TOP 60)
  :VIEW-SIZE #@(300 150)
  :VIEW-FONT '("Chicago" 12 :SRCOR :PLAIN)
  :VIEW-SUBVIEWS
  (LIST (MAKE-DIALOG-ITEM 'STATIC-TEXT-DIALOG-ITEM
           #@(70 17)
           #@(165 36)
           "Press the button to see the Listener greet you."
           NIL)
        (MAKE-DIALOG-ITEM 'BUTTON-DIALOG-ITEM
           #@(100 77)
           #@(104 22)
           "Press here"
           #'(LAMBDA
            (ITEM)
             ITEM
              (SETF YOUR-NAME
                (GET-STRING-FROM-USER
                   "Please type in your name."))
            (PRINT (FORMAT NIL "Hello, ~A!" YOUR-NAME)))
  :VIEW-FONT '("Monaco" 12 :SRCCOPY :PLAIN)
  :DEFAULT-BUTTON T)))
```

Menu code

Here is a code listing for the menu in Chapter 3:

```
(MAKE-INSTANCE 'MENU
  :MENU-TITLE "My Menu"
  :MENU-COLORS '(:MENU-TITLE 0)
  :MENU-ITEMS
  (LIST (MAKE-INSTANCE 'MENU-ITEM
          :MENU-ITEM-TITLE "Edit Menubar"
          :MENU-ITEM-ACTION
           #'(LAMBDA ()
             (IFT::EDIT-MENUBAR)))
        (MAKE-INSTANCE 'MENU-ITEM
          :MENU-ITEM-TITLE "That's a Fact!"
          :MENU-ITEM-ACTION
          #'(LAMBDA NIL
             (LABELS
                ((FACT (N)
                  (IF (EQ N 0) 1 (* N (FACT (- N 1))))))
                (PRINT (FACT (READ-FROM-STRING
                        (GET-STRING-FROM-USER
                          "Compute the factorial of:"))))))
        (MAKE-INSTANCE 'MENU-ITEM
          :MENU-ITEM-TITLE "Say hello"
          :MENU-ITEM-ACTION
           #'(LAMBDA NIL
               (MAKE-INSTANCE 'DIALOG
                 :WINDOW-TYPE :DOCUMENT
                 :WINDOW-TITLE "Good morning to you!"
                 :VIEW-POSITION '(:TOP 60)
                 :VIEW-SIZE 9830700
                 :VIEW-FONT '("Chicago" 12 :SRCOR :PLAIN)
                 :VIEW-SUBVIEWS
                   (LIST
                    (MAKE-DIALOG-ITEM 'STATIC-TEXT-DIALOG-ITEM
                     1114182
                     2359461
                     "Press the button to see the Listener greet you."
                     NIL)
```

(continued)

```
(MAKE-DIALOG-ITEM 'BUTTON-DIALOG-ITEM
 5046372
 1441896
 "Press here"
 #'(LAMBDA (ITEM)
    ITEM
    (SETF YOUR-NAME
      (GET-STRING-FROM-USER
         "Please type in your name."))
    (FORMAT T
            "Hello, ~A!" YOUR-NAME)))
 :VIEW-FONT '("Monaco" 12 :SRCCOPY :PLAIN)
 :DEFAULT-BUTTON T)))))))
```

Index

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and Microsoft Word software. Proof pages were created on the Apple LaserWriter printers. Final pages were created on the Varityper VT600 imagesetter. Line art was created using Adobe Illustrator. PostScript®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Palatino. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

Writer: Sarah Smith

Developmental Editor: Antonio Padial

Illustrator: Sandee Karr

Production Supervisor: Janet M. Anders

Special thanks to Gary Byers, Alice Hartley, David Lamkins, Steven Mitchell, William St. Clair, Andrew Shalit, and our skilled and helpful alpha and beta testers.