Digitool

For Macintosh Common Lisp versions 3.1 and 4.0

Getting Started with Macintosh Common Lisp

Contents

Figures and tables

Introduction

Contents

This introduction provides an overview of MCL, and describes the roadmap and resources for learning MCL.

Introducing Macintosh Common Lisp

Welcome to Macintosh Common Lisp, a fluid and flexible programming environment for developing software tools and applications.

Macintosh Common Lisp is built around a high-level object-oriented language, a fast, interactive compiler, a complete suite of programming tools, and a hassle-free object-oriented application framework.

■ Forty years of evolution have made Lisp both efficient and rich in programmer-oriented features. Its object system and large suite of libraries give it unparalleled power, while the presence of garbage collection and abstraction mechanisms ensure that you can easily apply that power to solve complex problems.

■ MCL's interactive programming environment saves time and simplifies debugging. You can compile, test, and correct functions and classes individually, without having to halt, recompile, relink and restart entire programs. Definitions can be recompiled and test functions can be executed in the context of the running program, allowing you to explore data structures and behaviors interactively. Typical recompilations take under a second.

■ MCL's programing tools begin with a fast fully programmable editor. Debugging is supported by a graphical inspector, source-code stepper, and stack backtrace. A single keystroke gives you access to signatures and documentation for built-in as well as user-defined objects.

■ The application framework provides major portions of the Macintosh Toolbox as high-level Lisp objects. The design makes it easy to interactively explore the application framework, quickly learning how to build a fully customized graphical user interface for your application. MCL also provides complete low-level access to all Macintosh OS calls and data structures, for those Macintosh features which are not yet supported by the framework.

MCL 4.0 and 3.1

This documentation describes two versions of Macintosh Common Lisp. Both are included in the standard software distribution.

Macintosh Common Lisp 4.0

Macintosh Common Lisp 4.0 runs as a native PowerPC application and includes a compiler that produces native PowerPC code.

Macintosh Common Lisp 3.1

Macintosh Common Lisp 3.1 runs as a native 68x application and includes a compiler that produces 68x code. It will also run under emulation on a Power Macintosh, although this configuration is no longer recommended.

For the most part, MCL 4.0 and MCL 3.1 provide the same interface and feature set. The same source code should compile and run with the same behavior (although at different speeds) in both versions. The same programming tools and libraries are available in both versions.

There are a small number of areas where MCL 4.0 and MCL 3.1 differ. Chief among these are the OS interface and foreign function interface. They also differ in their representation of objects and in the object runtime, so the behavior of the garbage collector, size limits of objects, etc. will vary between the two versions. Where MCL 4.0 and MCL 3.1 differ, the documentation will describe both, first describing the feature or behavior of MCL 4.0, and then describing the feature or behavior of MCL 3.1.

Roadmap: Learning MCL

Because of its interactive nature, it is relatively easy to learn how to use MCL and to quickly become productive using it.

Learning the Macintosh

Before using MCL, you should be familiar with at least the basics of the Macintosh: how to select menu commands, use windows, manipulate files, etc. If you have never used a Macintosh, you should take some time to familiarize yourself with it before proceeding to use MCL. This will not only make it easier to learn MCL, but will help ensure that the programs you write conform to Macintosh standards.

Learning Common Lisp

It is not necessary to know Common Lisp before beginning to use MCL. Indeed, MCL's interactive environment is a great way to learn Common Lisp. There are several ways to begin learning Common Lisp:

- A number of textbooks and tutorials are available for Common Lisp and the Common Lisp Object System (CLOS). Some are described in "Common Lisp Tutorials" on page 70.

- This Getting Started Guide describes some basic features of Common Lisp, and includes a short CLOS tutorial.

- The sample programs which come with MCL are a valuable source of information both on Common Lisp, and on the MCL application framework.

- Two Common Lisp reference works are available. These are not tutorials, but are complete descriptions of every feature of the language.

    Common Lisp: the Language, second edition is available in HTML format in the documentation folder of the MCL CD.

    The ANSI Common Lisp standard (X3.226-1994) is available in HTML format on the internet, at
    `<http://www.harlequin.com/books/HyperSpec/>`

MCL implements the language as described in *Common Lisp: the Language, second edition*. However, this version is close enough to the ANSI standard that the latter is still quite a useful reference when using MCL.

Learning MCL

This Getting Started Guide is the best way to begin learning MCL. It includes installation instructions followed by activity-driven introductions to the main elements of MCL:

- The read-eval-print loop, for interactively running Lisp code.
- The editor and listener windows.
- Compiling files.
- Aspects of the application framework.
- Debugging and on-line help tools.
- Customizing your environment.
- The Common Lisp Object System (CLOS)

The Macintosh Common Lisp Reference Manual provides a complete description of every feature of MCL, including the user interface, the application framework and other extensions which MCL provides to the Common Lisp language.

Online documentation is provided through balloon help and through documentation strings for built-in symbols.

Exploring MCL

By far the best way to learn MCL is just to dive in and explore. As you work through the Getting Started Guide, try out variations of the examples given, flip through the reference manual and try alternative functions. Because MCL is a fully type-checked and garbage-collected language, your explorations will be safe. Unless you are making direct OS calls, you won't have to worry about crashing. So sit back, relax, and explore.

For more information

A number of MCL-related resources are available through the Internet.

MCL Discussions and Announcements

There are active Internet discussions on all subjects relating to MCL. People use these discussions to ask questions, post tips, and share code. Engineers from Digitool monitor these discussions for questions.

The discussions can be accessed either as an e-mail mailing list, or through net news. Both these formats contain the same messages. Use the one which you find to be most convenient.

- `info-mcl@digitool.com`
  Subscribe to this list if you want to receive each posting to info-mcl as an individual e-mail message. To subscribe, send a message to `info-mcl-request@digitool.com`.

- `info-mcl-digest@digitool.com`
  Subscribe to this list if you want to receive a single e-mail message per day, containing all the day's messages from info-mcl. To subscribe, send mail to `info-mcl-digest-request@digitool.com`.

- `news:comp.lang.lisp.mcl`
  Read info-mcl through net news if you prefer this format to e-mail.

For those who do not want to participate in MCL discussions, but are still interested in hearing announcements relating to MCL, there is the announce-mcl mailing list.

- `announce-mcl@digitool.com`
  This mailing list is used to publish major announcements related to MCL. It is a very low-volume mailing list, so its guaranteed not to clutter your mail box. To subscribe, send mail to `announce-mcl-request@digitool.com`. It is not necessary to subscribe to both announce-mcl and info-mcl. Messages sent to announce-mcl are automatically routed to info-mcl as well.

Technical Support and Bug Reports

All technical support questions other than bug reports are handled via the `info-mcl` mailing list and are answered by Digitool's technical staff and by other members of the MCL community. Messages can be sent to `info-mcl@digitool.com` or posted to `comp.lang.lisp.mcl.`

Bug reports should be sent to `bug-mcl@digitool.com`. When you send in a bug report, please include a detailed description of your machine configuration and a description of your problem. If you can send a small fragment of code which reproduces the problem, that would also be of great help.

Other Internet Resources

The Digitool Web Page

The Digitool web page includes pointers to many MCL resources, including the mailing lists, the ftp site, lists of consultants who work in MCL, applications and research projects developed in MCL, price lists and product descriptions, etc.

The web page is located at `http://www.digitool.com/`

The MCL ftp Site

The MCL ftp site is home to patches and sample code from Digitool, and to MCL code contributed by users. If you have a utility or library which you'd like to share with the MCL community, you can upload it here. Conversely, if you're looking for a subsystem for MCL and think someone might have already written it, you can look for it here.

The MCL ftp site is located at `ftp://ftp.digitool.com/pub/mcl/`.

Contacting Digitool

Macintosh Common Lisp products are developed by Digitool, Inc. Please contact for additional information about Macintosh Common Lisp products, site licenses, and redistribution licenses.

Digitool, Inc.
One Main Street
Cambridge, MA 02142

Tel: 1-617-441-5000
Fax: 1-617-576-7680
E-mail: info@digitool.com
Web: http://www.digitool.com/

Chapter 1:

Installing MCL

Contents

This chapter tells you how to install MCL, and describes the Macintosh configurations required to run MCL.

System requirements

This section describes the system requirements for MCL 4.0 and 3.1, including the amount of disk space and RAM needed, and the versions of the Macintosh OS required.

MCL 4.0

Macintosh Common Lisp 4.0 will run on any PowerPC-based Macintosh. It requires System 7.5 or later.

A minimal installation requires approximately 20 MB of disk space. More space will be needed if you want to copy auxiliary material (such as the on-line documentation) from the CD to your hard disk. You can remove the "Examples" folder from the minimal installation to save about another 3 MB of disk space.

The memory requirements for running MCL 4.0 are described in detail in "MCL 4.0 Memory Requirements" on page 20. It is recomended that a Macintosh running MCL 4.0 have at least 16 MB of RAM.

MCL 3.1

Macintosh Common Lisp 3.1 will run on any 68x-based Macintosh. It is compatible with both System 6.x or 7.x, though versions System 7 are highly recommended and may be required in the future.

A minimal installation requires approximately 15 MB of disk space. More space will be needed if you want to copy auxiliary material (such as the on-line documentation) from the CD to your hard disk.

The memory requirements for running MCL will vary with the version of the Macintosh OS being used. It requires a minimum partition size of between 3.5 and 4 MB.

Installation

The MCL software comes on a CD. You can install it directly from the CD, or you can use the CD to create installation floppy disks for installation on machines which do not include a CD-ROM drive.

You can only perform a minimal installation from floppy disks.

Installing MCL 4.0

To install MCL 4.0 directly from the CD, copy the "MCL 4.0" folder to your hard disk. To install MCL 4.0 from floppies, create an MCL 4.0 folder on your hard disk by following the instructions for floppy disk installation on page 22. Once the MCL 4.0 folder is on your hard disk, move the following files and folders to their proper locations:

■ Aliases to the shared libraries "pmcl-kernel-4.0", "pmcl-library-4.0", and "pmcl-compiler-4.0" may be placed in your Extensions folder. This would allow your MCL-based applications to use them regardless of where the applications are located. Otherwise, these libraries or aliases to them must be in the same folder as the application when it is launched.

If you are working with the "Demo Version" of MCL 4.0, you need not worry about these shared libraries as they are built into the demo version.

■ Some of the files in the "Library" folder are referenced at runtime and/or compile-time by the MCL 4.0; by default, the "Library" folder is expected to be in the same folder as the MCL 4.0 application.

■ MCL 4.0 uses the Macintosh Thread Manager to support multiple processes. If your Macintosh does not have the Thread Manager installed, you should copy the file "ThreadsLib" from the MCL 4.0 distribution to your Extensions folder and reboot before running MCL 4.0.

IMPORTANT: Only install this version of the Thread Manager if your copy of the Macintosh OS does not already include it. System 7.5.3 includes the Thread Manager. To tell whether an older version of the Macintosh OS already has the Thread Manager installed, start up MCL 4.0 and select the Processes command from the Tools menu. If the dialog shows an "Initial" process but no "Listener" process, then your Macintosh does not have the Thread Manager installed. You should install it and reboot. **Checking your Extensions folder for the presence of a "ThreadsLib" file is not a reliable way to check for the presence of the thread manager, as it may be included in the System file.**

■ If you are upgrading from a previous version of MCL, you will need to recompile your fasl files.

MCL 4.0 Memory Requirements

The memory required to run MCL 4.0 will vary, depending on whether or not virtual memory is enabled. If virtual memory is enabled, it will run in a somewhat smaller memory partition.

With virtual memory enabled, MCL 4.0 will run in a minimal partition of 2.7 mb and be comfortable in a 4.8 mb partition. With virtual memory disabled, MCL 4.0 will run in a minimal partition of 3 mb and will run comfortably in a 5.1 mb partition. With virtual memory disabled, another 3.5 megabytes of MultiFinder memory must be available after MCL has been launched.

The reason for this difference is that the Mac OS "file-maps" static code and data when virtual memory is enabled. The number of bytes in this static code and data is effectively added on to the requested partition size. File-mapping will also be applied to programmer code and data in applications and Lisp environments created with `save-application`.

In System 7.5.3 with virtual memory enabled, the size requirements reported by the Finder are the amounts given in the `size` resource. With virtual memory disabled, the requirements reported by the Finder are the sum of the amounts given in the size resource, plus the size of static code and data.

The MCL 4.0 size resource is set for a generous partition showing approximately 4.8 mb with virtual memory and 5.1 mb without.

◆ **Important:** In starting up MCL 4.0, the Macintosh OS loads the MCL's shared libraries. If virtual memory is not turned on, these shared libraries require approximately 3.5 megabytes of free space in the Process Manager (Multifinder) zone. This is space that has *not* already been allocated to MCL. For this reason, when running without virtual memory you should not set MCL's partition size to take up all the remaining memory on your Macintosh. You should set its partition size such that at least 3.5 megabytes will be left free. If there is not enough memory for MCL 4.0 to load its libraries, MCL will display a dialog informing you of this. This also holds for the "Demo Version" of MCL 4.0, with the built-in libraries.

Installing MCL 3.1

To install MCL 3.1 directly from the CD, copy the "MCL 3.1" folder to your hard disk. To install MCL 3.1 from floppies, create an MCL 3.1 folder on your hard disk by following the instructions for floppy disk installation on page 22.

If you are using a 68030-based Macintosh without virtual memory, you should copy the file "PTable" from the MCL 3.1 folder to your Extensions folder and reboot your Macintosh. This system extension improves the performance of MCL's ephemeral garbage collector when running on 68030-based Macintoshes without virtual memory.

MCL 3.1 Memory Requirements

MCL 3.1 requires a memory partition of at least 3.9 MB. 5 MB or more is recommended.

Creating Installation Floppies

If you have access to a CD-ROM drive but it is not directly attached to the Macintosh on which you wish to install MCL, you install from floppy disks by using self-extracting archive segments provided on the MCL CD.

An installation of MCL 4.0 requires four disks and approximately 28 MB of free hard disk space. An installation of MCL 3.1 requires three disks and approximately 27 MB of free hard disk space.

The archive segments are located in the "MCL Floppy Disks" folder on the MCL CD. Each archive segment is stored in a separate folder within this folder. To create floppy installation disks, simply copy each archive segment folder to a floppy disk.

To install MCL on a hard disk using the newly created installation floppies, use the following steps:

1. Insert disk #1.
2. Double-click on the self-extracting archive application.
   After a brief pause, Compact Pro™ will ask you to show it the "Final Segment" of the archive.
3. Eject disk #1.
4. Insert the last disk.
5. Select the last self-extracting archive segment and click on "Load".
6. Compact Pro will ask where you want to put the MCL folder.
7. Select an appropriate location on your hard disk for Compact Pro to put the folder, then click on "Extract".
8. Insert each disk as Compact Pro requests it.

Disk #4 also contains a Compact Pro utility called "Extractor". Use this program if you want to get a few of the files out of the archive without extracting the entire environment.

Chapter 2:

A Brief Tour of MCL

Contents

This chapter introduces the Macintosh Common Lisp environment. In this chapter you'll create a simple program and learn how to use the basic tools of Macintosh Common Lisp.

Overview of MCL

MCL includes the best of traditional Lisp environments and of the Macintosh. Because of this, if you have experience with either the Macintosh or with other Lisp implementations, you'll likely feel immediately comfortable in MCL.

From the Macintosh side:

- You work with multiple windows, menus, and other standard UI elements.
- You use an editor that supports the standard Macintosh editing commands.

From the Lisp side:

- You write programs in Common Lisp and use the Common Lisp Object System (CLOS).
- You interact with your program through a Listener window.
- The editor is fully programmable and supports standard Emacs commands.
- At any time you can look at and edit the source code of any function, class, or other object. You can look up a documentation string for any definition.
- 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 fully programmable.

Starting MCL

If you followed the installation instructions in the previous chapter, you now have a full version of Macintosh Common Lisp in a folder on your hard disk. When you open this folder, inside you will see a set of other folders and the MCL application. The application will be called either "MCL 4.0" or "MCL 3.1," depending on which version you are using.

To start MCL, double-click on the MCL application. When MCL starts up, it loads the Lisp kernel and initializes the object system.

If a Lisp source-code file named "init.lisp" is present in the same folder as MCL, it is loaded. If a compiled init file is present ("init.pfsl" in MCL 4.0 or "init.fasl" in MCL 3.1), it will also be loaded. If both a source code file and a compiled file are present the newer one is loaded. You can use the init file to load customizations, preferences, additional libraries, and so on.

After MCL is finished starting up, you will see the MCL menubar at the top of the screen. In the middle of the screen you see a Listener window containing a welcome message.

Figure 2-1   The MCL Menubar
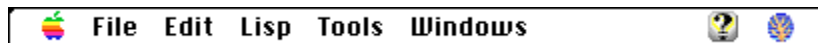
◆   **Note:** Starting up MCL 4.0 under virtual memory may take from several seconds to several minutes. The length of time depends on the model of machine and Macintosh OS version. Long delays are due to an interaction between the CFM subsystem and virtual memory subsystem of the Macintosh OS. Since this interaction occurs before any MCL code is run, we are unable to display a progress indicator or splash screen until it is complete.

Interacting with the Listener

When you are running MCL, the Lisp heap consists of a variety of objects: classes, functions, symbols, arrays, user-interface objects, etc. Functions and special forms are used to perform operations on these objects. Definitions are used to create and define the structure of these objects.

The Listener is a window into which you can type Lisp expressions for immediate execution. These expressions may create new objects and/or they may interact with existing objects. In either case, the result of executing the expression is printed as a return value in the Listener.

The Listener corresponds to a process in the MCL runtime. The process is in a loop, called a **read-eval-print loop**. The loop reads an expression, evaluates it, and then prints the result. In the case of MCL, evaluation usually consists of compiling the expression and then executing the result. However, the compiler is fast enough that the Listener still gives the impression of being an evaluator window.

There can be multiple Listeners in MCL. Each one will correspond to its own process. (Your program can also have additional processes which do not correspond to any Listener.) For now, we are just going to be using one Listener, and one process.

Figure 2-2 A Listener window

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 it appears in **boldface.** Text printed back appears in `normal type`. This convention is carried over to the examples given in this manual.

```
? "Hello, world!"        ;This is what you type.
"Hello, world!"          ;This is MCL's response.
```

Evaluating expressions

When you type a complete expression in the Listener and press Return, Macintosh Common Lisp immediately evaluates the expression and returns the result.

```
? (+ 10 20 30)
60
? (aref #(a b c d) 0)
A
? "Simple String"
"Simple String"
```

Every kind of Lisp expression has its own rules for evaluation. The first two examples given above are function calls, which evaluate to the result of calling the function on the arguments. The third example is a string constant. String constants (like other constants) evaluate to themselves.

Figure 2-3   A simple Listener interaction

```
┌─────────────────────────── Listener ────────────────────────────┐
│Welcome to Macintosh Common Lisp Version 4.0!                   ⬆ │
│? (+ 10 20 30)                                                    │
│60                                                              ▤ │
│? (aref #(a b c d) 0)                                             │
│A                                                                 │
│? "Simple String"                                                 │
│"Simple String"                                                   │
│?|                                                              ⬇ │
├──────────────────────────────────────────────────────────────────┤
│CL-USER|                                            ⬍ ⬅ ▥     ➡ 🗗 │
└──────────────────────────────────────────────────────────────────┘
```

You can also type more complex expressions into the Listener. Many expressions will contain nested subexpressions. It is also common to test out function and class definitions in the Listener.

```
? (+ 25 (* 10 10) (* 5 5))
150
? (defun factorial (n)
    (if (= n 0)
       1
       (* n (factorial (– n 1))))))
FACTORIAL
? (factorial 10)
3628800
```

➠ Before going on, try out using the Listener by typing in some simple expressions and seeing how they evaluate. (If you get into any error situations, you can always get out by typing command-period.)

When an evaluation is in progress, the About Macintosh Common Lisp command on the Apple menu is preceded by a diamond and the Listener minibuffer shows the word "busy."

Working with the Listener

The Listener is designed to make interacting with the Lisp environment very easy.

■ Pressing Return when the insertion point is not in the last line of text in the Listener causes the insertion point to move to the last line. If there is a selection, the selection is copied to the last line. If there is no selection, the text surrounding the insertion point moves down if it has been entered by the user, and is ready to execute.

■ Pressing Enter is equivalent to pressing Return twice. It performs both copy-down and execution.

- Pressing Control-Return causes a carriage return to be inserted. Use Control-Return when you want to reformat text in the Listener without performing copy-down or execution.

- Pressing Control-G creates a new input line without executing the current input line. The canceled input line is not erased, and you can use it later.

```
? Here is an input line  ;now press Control–G
?                        ;you have a new input line
```

- Pressing Option-G moves the previous input line to the bottom of the text in the Listener. Each time you press this key combination, a previous input line is moved to the bottom of the text in the Listener.

- Pressing Control-Option-P in the Listener moves the cursor to the previous input line. (Press the Control, Option, and P keys all at once to give this command.)

- Pressing Control-Option-N in the Listener moves the cursor to the next input line.

- The last result returned by the Listener is bound to the variable *. For example,
  ```
  ? (concatenate 'string "Hello" " Dolly")
  "Hello Dolly"
  ? (concatenate 'string "Well " *)
  "Well Hello Dolly"
  ```

- The Listener also supports all the standard Emacs commands supported by the MCL editor, as described in "The MCL editor, Fred" on page 30.

- Of course, standard Macintosh commands such as cut, copy, and paste also work in the Listener. And at any time you can save the text of the Listener into a text file.

- To get an online listing of Listener editing commands, press Control–question mark when you are in the Listener, or choose Listener Commands from the Tools menu.

The Listener and the Lisp Heap

You can close the Listener at any time. This has the effect of disposing of the text in the Listener. However, it does not undo any side-effects of the interactions you've had in the Listener. Those side-effects were performed on the global state of the Lisp heap, and remain even after the Listener they were entered in is closed.

Similarly, when you save a Listener, you are simply saving text. You are not saving a snapshot of the state of your current programming session. (It is possible to save snapshots with a different technique, as described in "Preserving programming sessions" on page 47.)

To create a new Listener at any time (whether or not there is already a Listener on the screen), choose New Listener from the File menu.

The following example shows that side-effects created through a Listener interaction persist, even after the Listener is closed.

➠ **Define a variable**

```
? (defvar new-variable 100)
NEW-VARIABLE
? new-variable
100                ;a variable evaluates to its value
? (+ new-variable 50)
150
```

➠ **Close the Listener window and create a new one.**

➠ **Use the variable in the new Listener window. It's still there.**

```
? (* new-variable new-variable)
10000
```

If you want to reset the Lisp heap and start over again, you do not do this by closing the Listener, but by quitting and restarting MCL.

The Listener and text files

The Listener is useful for immediate interactions with Lisp. However, it is not recomended for writing source code which you intend to keep. While it is possible to save the text of the Listener, this text includes not only your code, but also values returned by MCL, as well as additional printed information such as error messages and prompts.

You edit source code which you wish to keep in Fred windows. Fred is the MCL editor, described below. Fred stores source code as Macintosh text files with additional font information. Most source code is written in Fred windows, or written in the Listener and then copied into a Fred window for further editing into a source code file.

When you want to use the source code, you simply open the source file and edit or execute the code, as described in the following sections.

The MCL editor, Fred

Macintosh Common Lisp includes a powerful editor, Fred. Based on Emacs, Fred ("Fred Resembles Emacs Deliberately") includes the usual Macintosh editing features, such as multiple windows and mouse-based editing. In addition it includes a sophisticated set of features designed specifically for editing Lisp. Among these are parenthesis matching, smart indentation, and Lisp expression-oriented keyboard commands.

Moreover, because Fred is written in MCL, it is fully programmable: you can change its keymappings, change the behavior of existing commands, and add new commands.

In the next few sections, you will learn how to do the following:

- Create a new Fred window.
- Edit text in the window.
- Execute expressions in the window.
- Navigate through the text of the window.
- Save your work to a file on disk.

In doing this, you'll create a dialog box that asks your name and prints it out, along with a greeting.

Creating a Fred window

You create a Fred window by choosing the New or Open… command from the File menu.

➠ **Open a new Fred window.**

Choose the New command from the File menu.

You see two windows on the screen, your original Listener and a new Fred window. The new Fred window becomes the active window.

Figure 2-4   A Fred window and a Listener
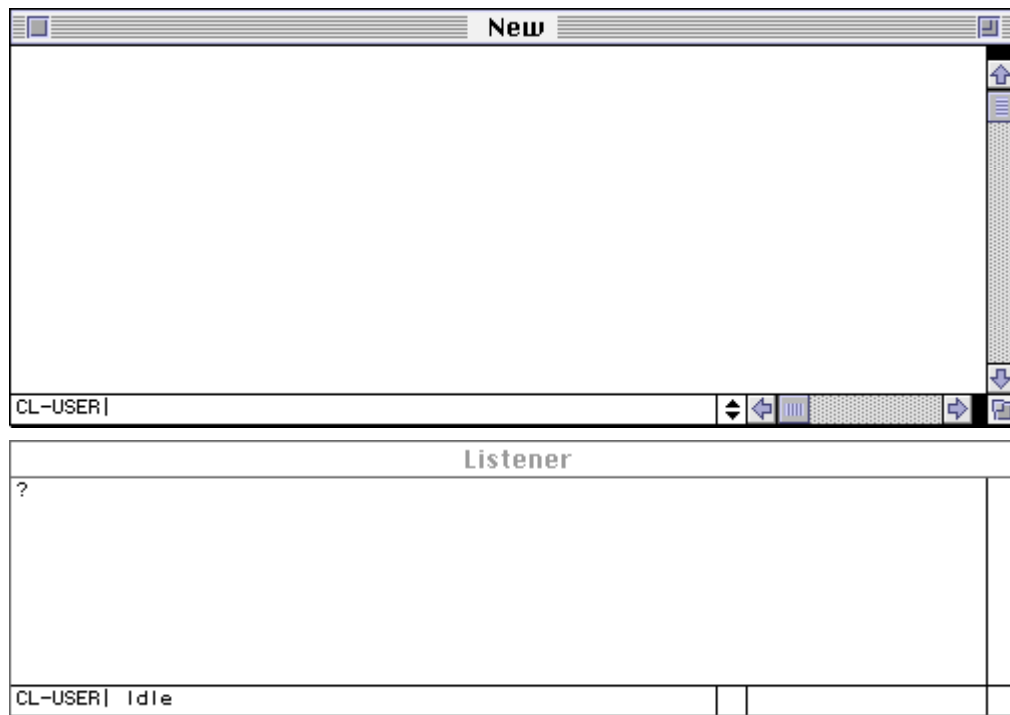
A Fred window displays an editor buffer, which contains a copy of the text you're editing. If the window is new, it doesn't yet correspond to a file on disk; the only copy is in the buffer. Otherwise the buffer contains a copy of the text from the file. When you save the window, its contents are copied to the file.

To enter text in your new Fred window, simply type.

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

Note that when you enter text in the window, a small cross appears in the title bar of the window. This indicates that the file has been modified since it was last saved.

Unlike text entered in the Listener, text entered in a Fred window is not immediately executed. However, you can easily execute the expression you have just typed.

Executing expressions in a Fred window

There are three ways to execute code from a Fred Window:

■ If you wish to execute all the expressions in the window, you can select the Execute All command from the Lisp menu.

■ If you wish to execute only a portion of the expressions in the window, you can select the portion and choose the Execute Selection command from the Lisp menu.

■ If you wish to execute a single expression, you can place the cursor at the start or end of the expression and choose the Execute Selection command from the Lisp menu, or press the Enter key.

When you execute code from a Fred window, the result of the last expression executed is printed in the Listener.

You can also execute all the code in a file without opening an editor window on the file. To do this, you load the file by choosing the Load command from the File menu.

➠ **Select the string `"Hello, world!".`**
The entire expression is highlighted.

➠ **Execute the expression by choosing "Execute Selection" or pressing Enter.**

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

Lisp-based editing

Fred includes a number of features that make it easier to read, edit, and navigate through Lisp code.

Matching delimiters

When the cursor is placed next to an expression delimiter such as a double-quote or a parenthesis, Fred will blink the matching delimiter, showing you the extent of the expression.

➠ **Place the cursor next to the closing double-quote of the string entered in the Fred window.**
Notice that the matching double-quote blinks.

➠ **Add the following code to the Fred window, below the string.**
```
(get-string-from-user
  "Please type in your name.")
```
Notice that when the cursor is at the closing parenthesis, the opening parenthesis blinks. If you move the cursor to the opening parenthesis, the closing one will blink.

You can navigate by expression using a number of different Fred commands. For example, pressing Control LeftArrow will move you to the left by one Lisp expression; pressing Control RightArrow will move you to the right by one Lisp expression.

➠ **Place the cursor at the end of the buffer, and press Control LeftArrow twice.**
The cursor should now be at the beginning of the buffer.

You can select an entire expression by double-clicking on one of the delimiters of the expression, or by pressing Control-Option Space when the cursor is next to one of the delimiters of an expression.

➠ **Position the cursor over the closing parenthesis and double-click.**
The entire expression is highlighted. If this were an expression within another expression, only the inner expression would be highlighted.

➠ **Execute the selected expression by choosing "Execute Selection" or pressing Enter.**
MCL executes the expression. The function `get-string-from-user` displays a dialog box with the specified prompt.

Figure 2-5   Executing an expression from a Fred window
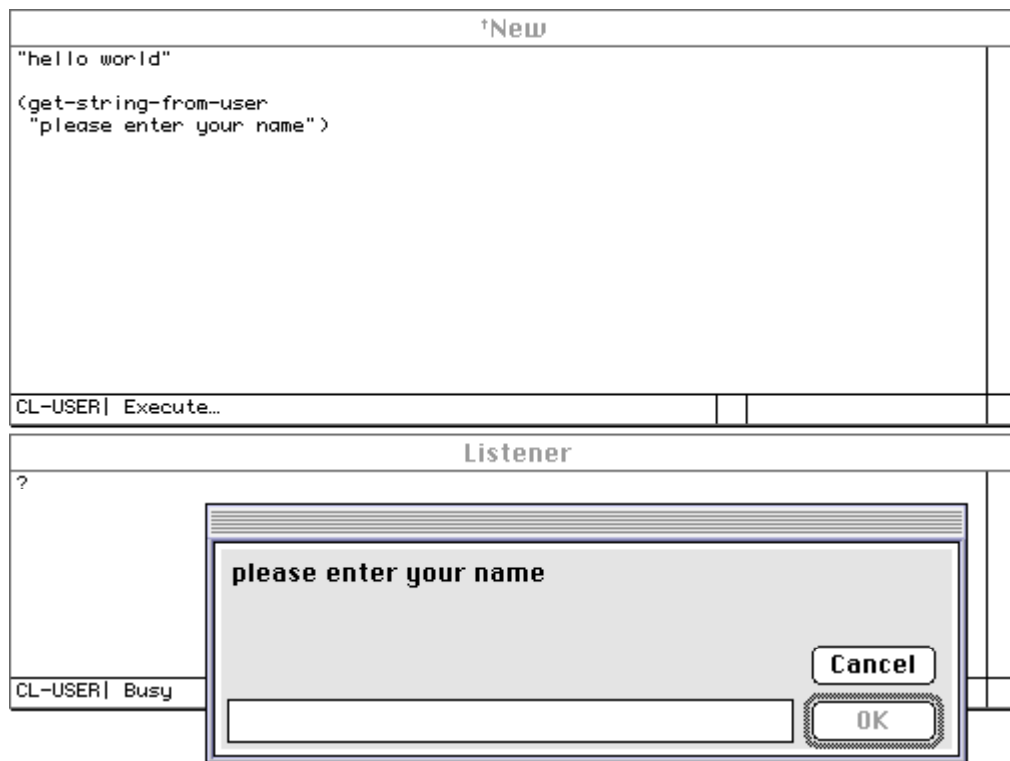
➠ **Type your name in the highlighted box and press Return.**
MCL returns your name as a string in the Listener window. This string is the result of the call to `get-string-from-user`.

Auto-indentation

Fred can automatically indent your Lisp source code for you.

➠ **Edit the call to `get-string-from-user` so that the value it returns is saved.**
Embed the call inside a variable definition:
```
(defvar my-name
  (get-string-from-user
   "please enter your name"))
```

Fred can auto-indent your code in one of three ways:

- If you press Control-Return rather than Return, Fred will auto-indent the new line.

- If you select a block of text and press Tab, Fred will auto-indent all the text in the selection.

- If you place the cursor on a single line and press Tab, Fred will auto-indent the line.

Cutting and pasting with Emacs commands

In addition to the standard Macintosh commands for cutting and pasting, Fred supports a number of Emacs commands for cutting and pasting. The Fred commands have two advantages over the Macintosh commands:

- They provide fine grain control over the text which is cut. For example, with a single keystroke you can cut a word, line, or expression.

- They store multiple pieces of cut text. Whereas the Macintosh clipboard can hold only a single item, the Fred kill-ring can hold any number of pieces of text.

➠ **Move to the start of the line containing `"Hello, world!"`.**

➠ **Press Control k**
Control k is "kill line". It "kills" (removes) text from the cursor to the end of the line and places it on the kill ring. Subsequent Control k's will kill following lines.

➠ **Type `(format nil`**

➠ **Press Control y**
Control y "yanks" (pastes) the top item from the kill ring into the editor window. The top line of the window should now read
```
(format nil "Hello, world!"
```

➠ **Continue editing.**
When you are done, the top line should read
```
(format nil "Hello ~a" my-name)
```

Executing your program

You now have a file with two expressions in it. It should look like this:

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


(format nil "Hello, ~A!" my-name)
```

Now try it out.

➠ **Execute the program.**
Choose the Execute All command from the Lisp menu.

Saving source code to a file

If you want to use this source code later, you must save it to a file.

➠ **Choose the Save command from the File menu or press Command-S.**
MCL will bring up the standard put-file dialog.

➠ Type the name "`Hello Dialog.Lisp`". To accept the name, either
press Return or click the Save button.

Getting help on Fred commands

As you have seen, there are a great number of Fred commands for a
great number of purposes. Fred commands are actually Lisp functions
bound to keystrokes. To help you find Fred commands you need, their
is a Fred Commands window which lists all these functions and their
corresponding keystrokes. To bring up the Fred Commands window
choose Fred Commands from the Tools menu or press Control-/ or
Control-? when the top window is a Fred window.

Figure 2-6   The Fred commands window
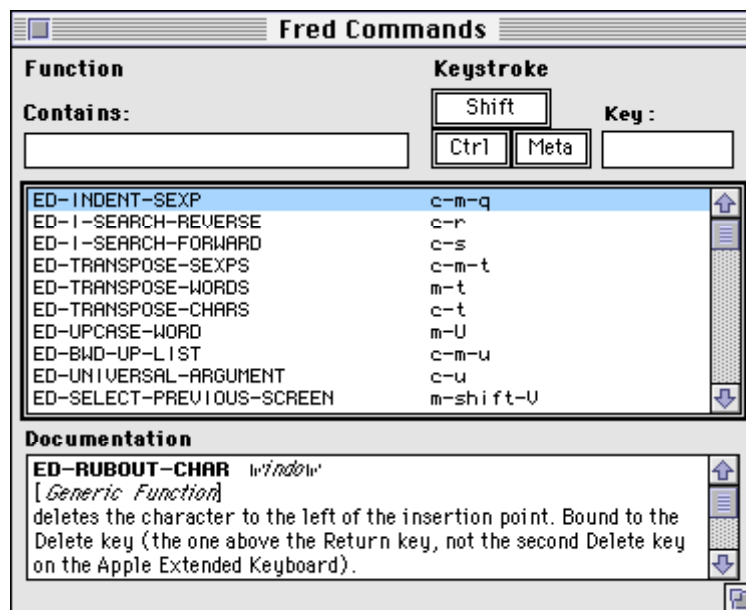
There are two ways to use the Fred Commands window. You can enter a likely substring of a command function name, and the window will display all the commands that include that substring in their name. Or, you can enter a keystroke or partial keystroke, and the window will display all the commands that use that keystroke or partial keystroke.

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

➠ **Choose Fred Commands from the Tools menu.**
The Fred Commands window appears.

➠ **Search for commands that contain the string "exp".**
Type "exp" into the text field in the upper-left of the window. MCL matches the string as you type it, character by character. The matching is not case-sensitive.

The scrolling list in the middle of the window displays all the commands which match the given criteria. On the left of the list is the name of the function which performs the command. On the right is the keystroke which invokes the command. In the right hand column, "c" stands for "control" and "m" stands for "meta" (the option key).

You can also narrow your search by keystroke.

➠ **Move the insertion point to the text field in the upper right of the Fred Commands window.**

➠ **Hold down the control key and the letter "a."**
Note that a much smaller number of commands are now shown: only those which contain "exp" in their command name, and which are invoked through a keystroke that includes control and "a."

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 standard Fred commands, and their command keystrokes. To show the Listener Commands window, choose Listener Commands from the Tools menu, or press Control-/ or Control-? when the top window is a Listener window.

Other Window Features

MCL has some extensions to the standard Macintosh window features:

■ Pressing Command-L makes the Listener the frontmost window.

■ Pressing the Option key and clicking the title bar of any window except a modal dialog makes that window the rearmost window.

■ Holding down the Option key when closing a window closes all the windows of that class.

■ Holding down the Control key when closing a window hides the window. Its title appears in italics in the Windows menu. Selecting the menu item shows and selects the hidden window.

■ Pressing Control-Option-L selects the second window. Pressing this keystroke repeatedly swaps the top two windows. This feature works only in Fred windows and in the Listener.

■ Pressing Control-Option-*number* followed by Control-Option-L selects the *numberth* window. This feature works only in Fred windows and in the Listener.

Compiling files

Earlier in this chapter we introduced the read-eval-print loop and explained that MCL actually compiles code entered into the Listener. Because the compiler is fast and you can recompile individual definitions, there is no noticeable delay.

Lisp source code is saved in text 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 the Compile File… command from the File menu. When you choose this command, MCL displays the choose-file dialog box, allowing you to choose 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 different extension. The extension for fasl files in MCL 4.0 is ".pfsl". The extension for fasl files in MCL 3.1 is ".fasl".

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

Just as you can recompile individual definitions, you can recompile individual files. You do not need to recompile all the files in your project when only one of them has changed.

File compilation example

We'll make a sample program a little bit larger, and then we'll compile it to a file.

➠ **Surround the call to `format` with a call to `message-dialog`.**
The resulting line of code should look like this:
```
(message-dialog (format nil "Hello, ~A!" my-name))
```

Notice that when you type the space after the name of the function "`message-dialog`", a message appears in the bottom of the Fred window. This message area is called the mini-buffer. When you type an open-parenthesis followed by the name of a function followed by a space, Fred displays the argument list of the function in the mini-buffer.

If you want to see what the function `message-dialog` does, you can get the documentation string for it. To do this, select the text "`message-dialog`" in the window, or place the cursor inside the text, and press Control-x Control-d. This is the MCL command for bringing up the documentation window.

Figure 2-7   The documentation window
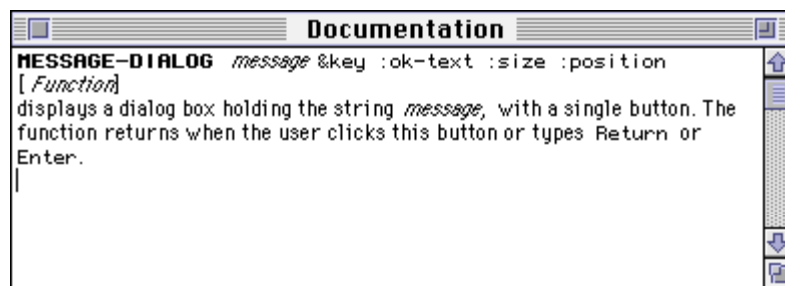
➡ **Replace the variable definition with a function definition, including a local variable binding.**
The resulting code will look like this:
```
(defun say-hello ()
  (let ((my-name (get-string-from-user
                   "Please type in your name.")))
       (message-dialog
        (format nil "Hello, ~A!" my-name))))
```

➡ **Compile the function.**
You do this by selecting it and choosing Execute Selection from the Lisp menu.

➡ **Now try out the function.**
Type `(say-hello)` in the Listener and press Return.

We've now defined a Lisp function which asks the user for their name, and then displays the name along with a greeting in a dialog box. If we want to use this function in a later Lisp session, we'll need to load it after restarting Lisp. To make that loading faster, we'll compile the file containing the function definition.

➡ **Save the file.**
Choose the Save command from the File menu.

➠ **Compile the file.**
Choose the Compile File… command from the File menu. MCL will prompt you to choose a file to compile. When you've chosen, it will prompt you to choose a name for the compiled file. Use the name it suggests for the compiled file.
MCL will then compile the file.

➠ **Quit and restart MCL**
Choose the Quit command from the File menu, or call the `quit` function from the Listener.

➠ **Load the file containing the function you've defined.**
Choose the Load from the File menu. You will be prompted to choose a file to load. You can choose either the source code file or the compiled file. In general, compiled-files load much more quickly (though with a file of this size, it doesn't make much difference).

➠ **Try out the function, to make sure it loaded properly.**
Type `(say-hello)` in the Listener and press Return.

➠ **Go back to the source code of the function.**
Position the cursor inside the text "`say-hello`" and press Option-Period. MCL will open the source code file containing the definition (if it is not already open) and it will scroll to the definition.

What you've learned

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 executed code in it. Then you added a Fred window and edited source code in it. You created a small program in the window, executed it, and saved it to a file. In writing this program, you've learned how to put up simple message and prompt windows, and how to format a simple message. You've used Fred commands to navigate, indent, cut and paste text, find documentation of functions, and go to the definition of functions. Finally, you've compiled a file of source code, and loaded a compiled file.

Chapter 3:

The Application Framework

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.

In this chapter, you'll learn how to create windows, dialog-items, menus, and menu-items. You'll also learn how to save snapshots of a Lisp session.

Overview

Macintosh Common Lisp includes a set of class libraries which provide high-level access to the Macintosh user interface. The classes and methods in these libraries make it easy to create an interface for your program without worrying about the low-level details of system calls. Most importantly, the class libraries are safe. They provide crash-free access to Macintosh UI.

In the last chapter, we used two built-in dialog boxes: a message dialog and a string-query dialog. In this chapter, we'll show how to build your own dialogs and also how to put up menus. We'll also describe some ways to customize your MCL environment.

Complete details on creating windows, menus, and other UI elements are given in the MCL reference.

Windows and views

MCL provides a set of classes for displaying windows, dialog boxes, and common items in windows such as static-text dialog items, editable-text dialog items, buttons, check boxes and radio buttons.

Creating a window

It's easy to create windows in MCL.

➠ **Create a window.**
Enter the following code in the Listener, and press Return.
```
(make-instance 'window)
```

➠ **Set the window's title.**
Enter the following code in the Listener, and press Return.
```
(set-window-title (target) "Echo")
```

The function `target` returns the second window. It makes it easy to use the Listener to operate on another window.

➠ **Set the window's size.**
Enter the following code in the Listener, and press Return.
```
(set-view-size (target) #@(240 180))
```

Most objects which are displayed on the screen are views. In particular, windows and items displayed in windows are views. One property shared by all views is a size. Because of this, we use the function `set-view-size` to set the size of a window. The window class has inherited this function from the view class. The syntax `#@(`*i1 i2*`)` is used to indicate point literals. In this case, we are saying the width (or horizontal coordinate) should be 240, while the height (or vertical coordinate) should be 180.

➠ **Check the window's title.**
Enter the following text in the Listener, and press Return.
```
(window-title (target))
```

Just as there are functions for setting the attributes of windows and other UI objects, there are functions for retrieving their attributes.

Window init-keywords

We've seen that it's possible to interactively create, modify, and inspect a window interactively. However, most of the time you'll want to create a window with the chosen attributes preset. You can do that by specifying the attributes as keyword arguments to `make-instance` when you create the window.

➠ **Create a window with preset attributes.**
Enter the following text in the Listener, and press Return.
```
(make-instance 'window
  :window-title "Echo"
  :view-size #@(240 180))
```

Views and subviews

The items displayed inside a window are the **subviews** of the window. In the simplest cases, there is only one level to this view hierarchy: the window is the outermost view, and it contains subviews. However, it is possible to have deeper levels of nesting. Subviews in the window can, in turn, have their own subviews, etc.

Events such as mouse clicks and keystrokes are handled by the window, which often passes control on to an appropriate subview.

Adding a button

➧ **Add a subview to the window.**
Enter the following text in the Listener, and press Return.
```
(add-subviews (target)
                 (make-instance 'button-dialog-item
                   :dialog-item-text "Beep"
                   :dialog-item-action
                   #'(lambda (self)
                        (declare (ignore self))
                        (beep))))
```

Notice that a button has been added to the window, with the specified text. When you press the mouse on the button, the button highlights, and tracks the mouse movement appropriately. When you release the button, the action is run.

In this case, the action is an anonymous function. That's the meaning of the expression beginning with "`#'(lambda`…". The lambda expression creates a function of one argument, `self`, it ignores that argument, and it then calls the function `beep`. The anonymous function is stored as the action-function of the dialog item. It is called when the button is clicked.

When a button's action-function is called, the button itself is passed as an argument. That way the function can find out what dialog it is in, etc. In this case, we do not need to use that information. That's why we declare that we are ignoring the argument. If the lambda expression did not include that declaration, the compiler would issue a warning about an unused argument.

Adding an editable text item

Another common type of subview is an item that allows the user to enter and edit text.

➧ **Add an editable-text item to the window.**
Enter the following text in the Listener, and press Return.
```
(add-subviews (target)
                 (make-instance
                   'editable-text-dialog-item
                   :view-size #@(160 16)
                   :view-nick-name 'visitor))
```

Now if you select the window, you can edit text in the newly added subview.

When we created the editable-text dialog item, we specified a `view-nick-name` for it. This allows us to locate the view later, if we want to perform some operation on it.

➠ **Set the font of the editable-text item.**
Enter the following text in the Listener, and press Return.
```
(set-view-font (view-named 'visitor (target))
               "New York")
```
Now when you type in the editable-text item, the typing will be displayed in the font "New York", rather than the default "Chicago" font.

Retrieving the text from an editable text item

An editable-text dialog item is only useful if there is a way to retrieve the text that has been entered into it. We'll now add another button to our window which does just that.

➠ **Add a text-retrieval button to the window.**
Enter the following text in the Listener, and press Return.
```
(add-subviews (target)
               (make-instance 'button-dialog-item
                 :dialog-item-text "Echo"
                 :dialog-item-action
       #'(lambda (button)
            (let* ((text-item (view-named 'visitor button))
                   (text (dialog-item-text text-item)))
              (format t text)))))
```

Creating a complex window

As noted above, it is possible to specify the attributes of a window when the window is created. The subviews can be part of these attributes. Combining the snippets of code we created in the Listener into a single function, we get the following. When this function is called, it displays a window containing a button and an editable-text dialog item. When the button is pressed, it retrieves the text from the editable-text dialog item and displays it in the Listener.
```
(defun make-echo-window ()
  (make-instance 'window
    :window-title "Echo"
    :view-size #@(240 40)
    :view-subviews
    (list
     (make-instance 'button-dialog-item
       :dialog-item-text "Echo"
       :dialog-item-action
       #'(lambda (button)
            (let* ((text-item (view-named 'visitor button))
```

```
                (text (dialog-item-text text-item)))
              (format t text))))
       (make-instance 'editable-text-dialog-item
         :view-size #@(160 16)
         :view-nick-name 'visitor))))
```

➠ **Define and test the function.**
Create a new Fred window, enter the definition in the window, save
the window, and compile the function. Then try calling the function
from the Listener.

Menus

Just as it's easy to create windows and views with MCL, it's also easy to
create menus and menu-items.

You may want to create menus and menu-items as part of creating an
entirely new menubar for your application. You may also want to add
new menus to the existing menubar, or add new menu-items to existing
menus. This is one way to customize your MCL environment. Because
MCL's built-in menus are written in MCL, they behave in just the same
way as any menus you might add.

Adding a menu and a menu-item

To add a menu to the menubar, we create an instance of the class menu,
and install it. Because we will want to perform other operations on this
menu, we will store it in a variable when we create it.

➠ **Create a menu and add it to the menubar.**
Execute the following expressions in the Listener:
```
(defvar *custom-menu*
  (make-instance 'menu
    :menu-title "Custom"))
(menu-install *custom-menu*)
```

➠ **Add a menu-item to the new menu.**
Execute the following expression in the Listener:
```
(add-menu-items
 *custom-menu*
 (make-instance 'menu-item
   :menu-item-title "Echo Window"
   :menu-item-action 'make-echo-window))
```

That's all there is to it. You've now added a menu and a menu-item to your Lisp environment. Try selecting the menu-item, to see that it works.

The interface toolkit

In the previous sections, you created windows and menus by writing the code to generate them by hand. It is not always necessary or desirable to do so.

MCL comes with a tool for creating user interfaces graphically. It is called the "Interface Toolkit", or "IFT" for short. It comes in source code form, with instructions, and is located on your MCL CD. It allows you to build windows, subviews of windows, menus and menu-items graphically, and then generates the source code for these UI elements automatically. It can even be extended to handle new kinds of UI elements that you design.

Preserving programming sessions

When you program in MCL, you are creating objects: functions, classes, and other instances. You are building up state in your Lisp environment.

At times it is desirable to save a snapshot of your environment for later resumption. It may be that reproducing a particular state is very time consuming, either because it involves loading many files of Lisp code, because it requires performing other lengthy computations, or perhaps because your environment has gotten into an anomalous state which you want to examine later.

MCL includes a facility for saving snapshots—also known as images—of a running Lisp session. These snapshots can be restarted at a later time. They start up just about as quickly as starting up a new Lisp system.

To save an image, just call the function `save-application`. This function takes one required argument and a number of keyword arguments. The required argument is a pathname. It closes any open windows (these cannot be saved), writes an image to disk at the selected pathname, and then exits to the Finder. The saved image is a stand alone application. It can be restarted by double-clicking it, just like any other stand alone application.

◆ **Note:** To restart an image created in MCL 4.0, the image must have access to MCL's three libraries, as described in the installation instructions. These libraries must be in the same folder as the image, or they must be in the Extensions folder, or aliases to them must be in one of these folders.

◆ **Note:** If you save an image with the same signature as MCL (the default), then double-clicking on an MCL file in the Finder may launch the new image, or it may launch MCL. The choice is made by a caching mechanism in the Finder. To avoid confusion, you may wish to save images with a different signature, or else not launch MCL by double-clicking MCL files.

A complete description of `save-application` is given in the MCL Reference.

The functionality of `save-application` is also available through the Save Application… and the Extensions/Save Image… commands on the Tools menu.

What you've learned

You've created a window and interactively added dialog-items to it. You saw how all the steps of initializing a window can be performed when the window is first created. You customized your environment by adding a menu with menu-item to the menubar. Finally, you learned how to save out Lisp images, to preserve entire programming sessions for later resumption.

Chapter 4:

Debugging

Contents

This chapter describes some of the facilities in MCL for accessing object information and documentation, inspecting objects, and debugging programs and processes.

MCL's multiple debugging facilities

Macintosh Common Lisp provides many ways for you to examine and debug functions, and source code, and to inspect the state of your Lisp system:

■ **Documentation commands.**
A set of Lisp functions and Fred commands give you access to the source code, documentation strings, and argument lists of functions and other objects.

■ **Introspection commands.**
A set of Lisp functions, Fred commands, and menu commands provide information on the state of your Lisp system (such as the amount of room available) and allow you to inspect objects and follow links between objects.

■ **Debugging tools.**
A set of tools allow you to monitor functions and single-step through functions, examine stack backtraces, and correct programs and restart from error situations.

Documentation commands

The documentation commands give you access to source code, argument lists, and documentation strings for your definitions as well as many built-in definitions.

Source code

When definitions are compiled and the value of the global variable `*record-source-file*` is true, the source file from which the definition was compiled is remembered. You can later navigate from the symbol that was defined (i.e. the class name, function name, variable name, etc.) to the source code.

The default value of `*record-source-file*` is true.

There are three ways to retrieve the source code of a definition:

■ By using the function `edit-definition`, which takes a symbol as its first argument.

- By using the Fred command Option-Period, when the cursor is in a symbol or when a symbol is selected. (This command is often called "meta-point" or "meta-dot.")

- Through the inspector, the stack backtrace, and other graphical tools.

In all cases, if the source code information of the definition was recorded, the source code file will be opened and scrolled to the definition.

In some cases, there will be multiple definitions for a single symbol. For example, there may be both a class and a function defined with the same name, or there may be many method definitions on the same generic function. In these cases, you are presented with a list of definitions and prompted to choose one.

Argument lists

You can retrieve the argument lists of compiled functions. Depending on how the functions were compiled and loaded into the Lisp environment, these argument lists may or may not contain the original argument names. If they do not contain the original names, they will still show the correct number and types of arguments, along with automatically generated names.

When working interactively, the names of arguments will be remembered if the value of the global variable `*save-local-symbols*` or of the global variable `*save-definitions*` is true. When loading functions from fasl files, the names of arguments will be remembered if the value of the global variable `*fasl-save-local-symbols*` or of the global variable `*fasl-save-definitions*` was true when the fasl files were compiled. The default value of all these variables is false.

There are four ways to retrieve the argument list of a function:

- By using the function `arglist`, which takes a symbol as its first argument. This function returns the argument list of its argument, and also returns a second value describing how it computed the argument list.

- By using the Fred command Control-x a, when the cursor is in a symbol or when a symbol is selected. This will cause the argument list of the function associated with the symbol to be displayed in the minibuffer of the current Fred window.

- If `*arglist-on-space*` is true, then typing an open parenthesis followed by the name of a function followed by a space will cause the argument list of the function to be displayed in the minibuffer of the current Fred window. The default value of `*arglist-on-space*` is true.

- The inspector displays the argument lists of functions when they are inspected.

Documentation

Many built-in definitions have associated documentation. In addition, your own definitions can include documentation strings. These documentation strings will be returned when the definition is compiled only if the value of the global variable `*save-doc-strings*` is true. The default value of this variable is false.

The following function definition includes a documentation string. The exact placement of the documentation string varies among different types of definitions. Check the Common Lisp specification for details.

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

There are three ways to retrieve the documentation of a definition:

- Through the function `documentation`, which takes a symbol as its first argument and a documentation type as its second argument. It returns the documentation string of that type for the symbol if one exists. The second argument should be `function` or `type`. See the Common Lisp documentation for details.

- By using the Fred command Control-x Control-d, when the cursor is in a symbol or when a symbol is selected. This will cause the documentation associated with the symbol to be displayed in a documentation window.

- Through the Get Info window or the Inspector, available through the Tools menu.

Introspection commands

A set of Lisp functions, Fred commands, and menu commands provide information on the state of your Lisp system (such as the amount of room available) and allow you to inspect objects and follow links between objects.

Free space

The Common Lisp function `room` prints out information describing the free space available in Lisp. This function takes an optional argument, true or false, which can be used to control the level of detail in the space report.

Finding symbols

The apropos facility is used to search through the space of symbols for symbols which contain a particular substring. This function is available both through a Common Lisp function and as a menu command.

The apropos function

The Common Lisp function `apropos` takes a symbol or string as its argument and prints out all symbols known to MCL that contain the symbol or string. A package may be supplied as an optional second argument. If supplied, only symbols from the specified package are included in the print-out.

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 'subview)
           ADD-SUBVIEWS, Def: STANDARD-GENERIC-FUNCTION
           DO-SUBVIEWS, Def: MACRO FUNCTION
           MAP-SUBVIEWS, Def: STANDARD-GENERIC-FUNCTION
          :ORDERED-SUBVIEWS,  Value: :ORDERED-SUBVIEWS
     CCL::ORDERED-SUBVIEWS, Def: STANDARD-GENERIC-FUNCTION
     CCL::REMOVE-ORDERED-SUBVIEW, Def: FUNCTION
           REMOVE-SUBVIEWS, Def: STANDARD-GENERIC-FUNCTION
           SUBVIEW
           SUBVIEWS, Def: STANDARD-GENERIC-FUNCTION
          :VIEW-SUBVIEWS,  Value: :VIEW-SUBVIEWS
           VIEW-SUBVIEWS, Def: STANDARD-GENERIC-FUNCTION
```

The apropos window

The apropos window is available through the Apropos… command on
the Tools menu.

Figure 4-1   The apropos window
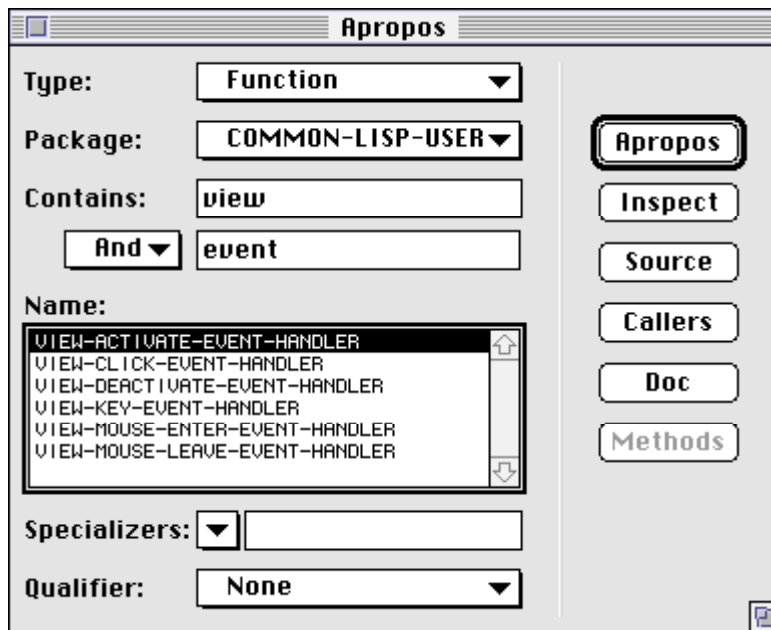
The apropos window provides access to the `apropos` function with a
number of additional options. You can restrict the types of symbols
returned (i.e. only symbols with function definitions, class definitions,
etc. will be returned, if so specified); you can specify two strings and
only view symbols which contain both, either of, or one but not the
other of the strings.

Once you have found a symbol, you can select it and then perform a number of different actions on it, such as inspecting it, locating its source code, documentation string, etc. If you are performing one of these actions on a generic function, you can redirect it to a method on the generic function by choosing the specializers and/or qualifiers of the method you wish to operate on.

Examining objects with the Inspector

The Inspector lets you look quickly at any component of an object. It provides a route to the source code of the object. It even lets you edit many of the attributes of an object that is being inspected. The Inspector is readily available from many other windows such as the Apropos window.

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 your Macintosh 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.

There are a number of ways to invoke the inspector:

- You can inspect a symbol by selecting it or placing the cursor inside it and issuing the Fred command Control-x Control-i. When you inspect a symbol, the inspector will also show you all the objects associated with the definitions of the symbol, such as the function if the symbol has a function definition, or the class if the symbol has a class definition.

- You can use the Common Lisp function `inspect`. This function takes any object as its argument, and inspects that object.

- The Inspector command on the Tools menu has a number of sub-commands for inspecting some built-in aspects of MCL.

- Many of the other debugging tools, such as the apropos window or the stack backtrace (described below) allow you to inspect the objects they display simply by double-clicking them.

The following sections show some examples of using the Inspector.

Inspecting an object with `inspect`

Following is an example of inspecting a window w that has already been closed.

➥ **First create a window.**
```
? (setq w (make-instance 'window))
#<WINDOW "Untitled" #x583621>
```

➥ **Then close it.**
```
? (window-close w)
NIL
```

➥ **Inspect it.**
```
? (inspect w)
#<INSPECTOR-WINDOW "#<WINDOW  #x583621>" #x452511>
```

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

Figure 4-2   An Inspector window showing components of a window
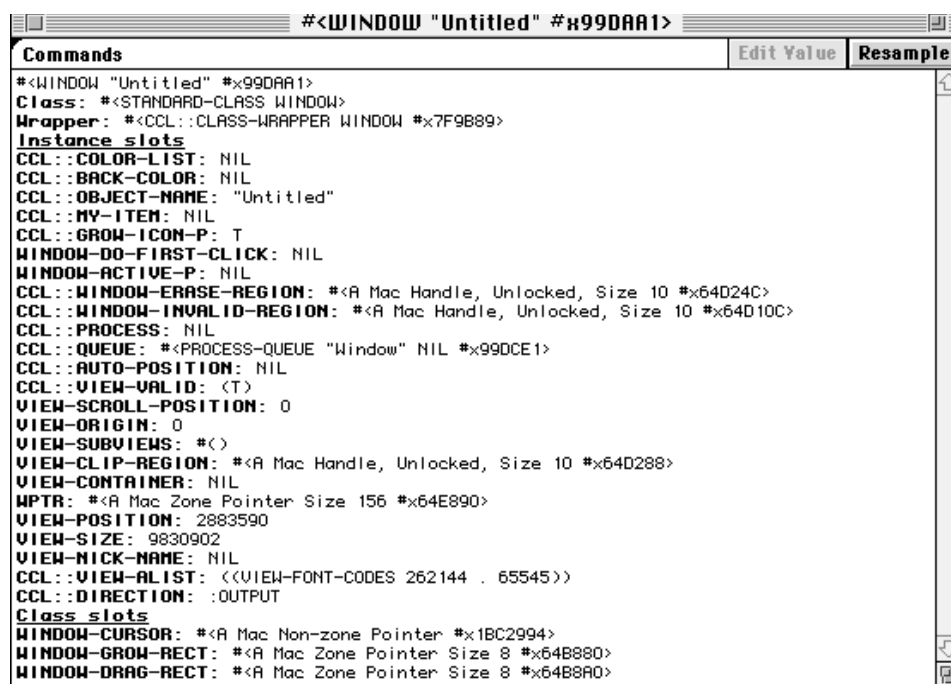
You can inspect this window and its components. For example, you could inspect the class of the window, `#<STANDARD-CLASS WINDOW>`.

Inspecting objects from other tool windows

You can inspect objects from many of the other tool windows. For example, from the apropos window you can double-click a symbol to inspect it. You can then double-click any of the symbol's components to inspect that component.

Figure 4-3 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 an Inspector window showing 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.

The user double-clicks the package `#<PACKAGE "CCL">` to inspect that package.

Figure 4-3   An apropos window and the Inspector
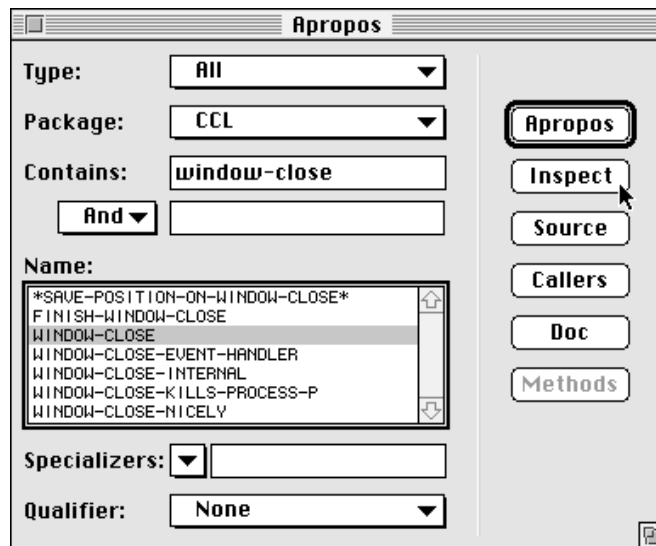
Double-clicking on `window-close` brings up an Inspector window for that symbol:
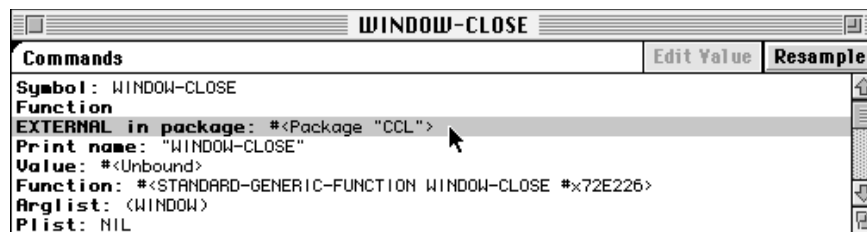
Double-clicking on #<Package "CCL"> inspects the package:
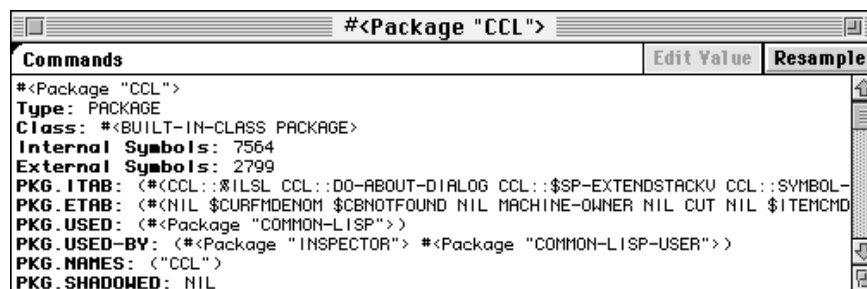
Errors and Break Loops

When Macintosh Common Lisp signals an error, it prints an error message. The error message tells you what kind of error occurred, the function in which it occurred, what objects were involved in the error, and what your options are for recovering.

The details of the error message are specified by the person who wrote the code which detected and signaled the error. Similarly, the recovery options are put into place by the author of the software system in which the error occurred. If there are no recovery options, there is always the option of simply canceling the computation.

Some recovery options may be listed in the error message itself. Others are given in the Restarts window, which is accessed through the Restarts… command on the Lisp menu.

Reading an error message

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

➠ **Enter an expression which contains an 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 there is no such function as `close-window`. The function `toplevel-eval` (which is part of the read-eval-print loop) gets an error when trying to call this non-existent function.

The error message first gives the nature of the problem.

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

The message notes the function in which the error occurred:

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

Next the message describes your options for recovery or cancellation. In this case, you can either retry calling the function by choosing Continue, or you can cancel the entire computation with Abort. As always, you can check the Restarts window for additional options. (They keystrokes command-/ and command-. are shortcuts for the Continue and Abort commands on the Lisp menu.)

```
> 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 or aborting

When an error occurs, you can always cancel the execution and often you can continue.

Aborting

When the problem is obvious and is not inside a deeply nested time-consuming computation, it is often easiest to cancel out of the execution and start over.

➠ **Cancel out of the execution.**
Press Command-period or choose the Abort command from the Lisp menu.

➠ **Edit the form.**
Press Option-G to bring back the last expression as you originally typed it. Edit the expression to replace the call to `close-window` with a call to `window-close`. (In the more usual case, the error would be in a compiled function or other definition. You would use option-period to get to the source code, and then edit and recompile the definition.)

➠ **Retry the execution.**
Press Return to try again. Assuming no other errors have been introduced, execution should complete successfully and the window should close.

Recovering

When an error occurs in the middle of a long computation—for example, in the middle of a 3,000 file compilation—you will probably want to find a way to recover and continue the computation rather than aborting and starting over.

Options for recovery may include skipping some part of the computation (for example, not compiling one of the 3,000 files), or they may include changing your Lisp environment in some way so that the function which hit the error can be reinvoked successfully.

The simple error above can be handled by defining the function `close-window` and retrying the computation:

➠ **Recreate the error condition by creating a window and trying to close it by calling `close-window`.**

➠ **Define `close-window`.**
```
1 > (defun close-window (w) (window-close w))
CLOSE-WINDOW
```

➠ **Continue from the error.**
Press command-/ or choose Continue from the Lisp menu. The execution should complete successfully and the window should close.

The restarts window

The Restarts window automatically gathers all possible ways of recovering from an error. It always offers you the option of canceling and sometimes offers other options for 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 shown above, Restarts offers more specific suggestions. Here the option `"Apply specified function to (#<WINDOW "Untitled" #x4D4D99>)..."` lets you call another function, rather than `window-close`.

Figure 4-4   The restarts window
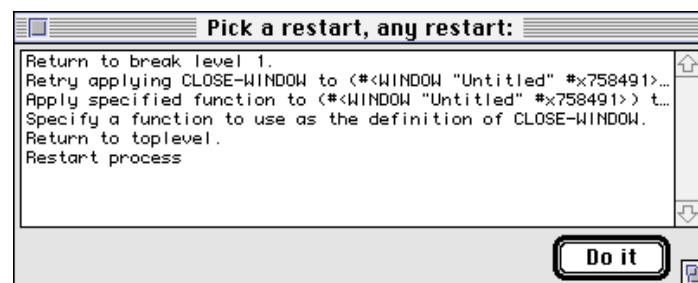
➠ **Recreate the same error as above.**

➠ **Open the Restarts window.**
Select the Restarts… command from the Lisp menu. Notice that when you attempt to call a function which is not defined, one of your options is to complete the computation by calling a different function on the same arguments.

➠ **Find the correct function to call.**
Select the Apropos command from the Tools menu. Search for all functions that include both "close" and "window" in their names. There are only two such functions, and only one likely candidate for this operation.

➠ **Restart the computation.**
Return to the Restarts window and select "Apply specified function…". Enter the correct function name in the ensuing dialog, and press Return. MCL completes the computation and closes the window.

Since the Restarts window is generated by a fixed algorithm, it often suggests the same restart multiple times. In such cases, it doesn't matter which copy you choose. (However, if there are multiple levels of break loop, make sure you're not choosing to return to a point in the wrong one!)

The break loop

The break loop is a read-eval-print loop that is activated when an error occurs or when you explicitly call the function `break`. The process that was running is suspended, and a new process—the break loop—is activated. A break loop acts like the normal read-eval-print loop except that it runs on top of your suspended program, allowing you to interact with MCL on top of your program. From a break loop you can examine your program state (including the stack), make changes, and then either resume or cancel the operation of your program.

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 4-5).

Figure 4-5   Effects on the stack of break, abort, and continue

read-eval-print loop
User code in execution

abort →

read-eval-print loop

read-eval-print loop
User code in execution

break →

read-eval-print loop
User code suspended
break loop
New user code in execution

continue →

read-eval-print loop
User code resumed
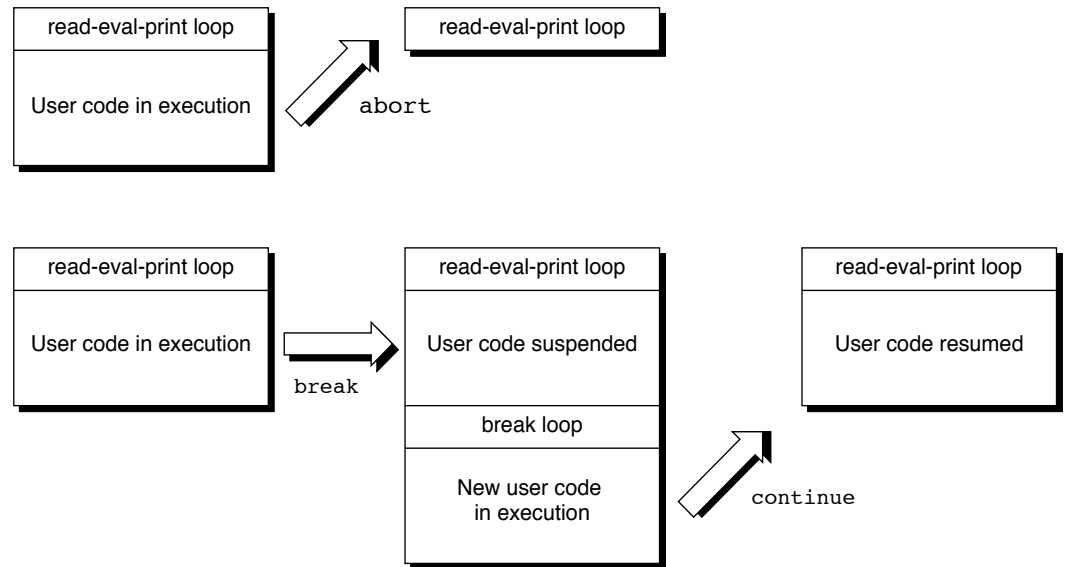
Break loops add new areas to the stack, whereas Abort and Continue remove areas from the stack. Within a break loop, the normal question mark prompt is replaced by a number and an angle bracket. The number of the prompt represents the level of the break loop.

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. You can also edit the values of local variables, though by doing so you risk corrupting your Lisp runtime.

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

The stack backtrace

Whenever Macintosh Common Lisp is in a break loop, the stack backtrace is enabled, letting you examine the functions and values on the stack of the suspended process. If the value of the global variable `*backtrace-on-break*` is true, the stack backtrace window opens automatically when MCL enters a break loop. If the value of `*backtrace-on-break*` is false, you can bring up the stack backtrace window by choosing the Backtrace command from the Tools menu.

The stack backtrace lists all the functions awaiting return values on the stack at the time of the error. (Note, because of tail recursion optimization, this may not be all the functions that were called.) 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.

Figure 4-6   A stack backtrace
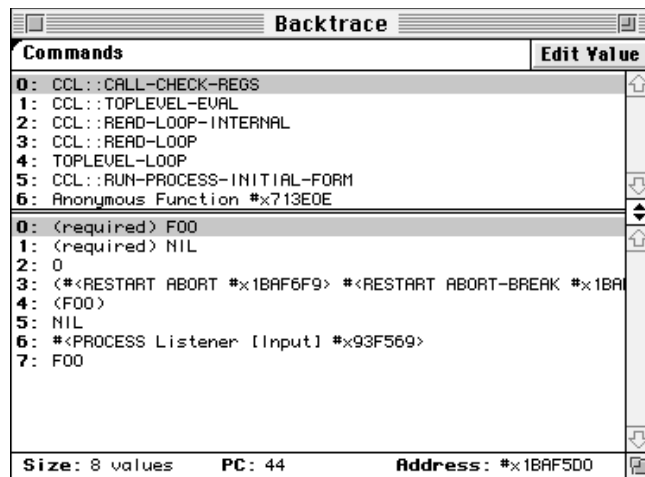
There are two tables in the Stack Backtrace window (Figure 4-6). The upper table shows you the functions pending on the stack. You can examine the stack frame of a function by selecting the function. This shows you the values of local variables active in the function, including its arguments. It also shows you any special variables that are bound within the function. You can inspect a function by double-clicking it.

The lower table shows the stack frame of the function that is selected in the upper table. The names of parameters and local variables will be shown if the corresponding function was compiled interactively if when *save-local-symbols* was true or was loaded from a fasl file that was created when *fasl-save-local-symbols* was true; otherwise automatically generated names will be used.

You can inspect an item in the lower table by double-clicking it.

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. By comparing the program counter to a disassembly of the function (available in the inspector), you can determine where in the function execution halted.

The command pop-up in the stack backtrace lets you go to the source code of a function, invoke recovery options, inspect values, and edit values.

Processes

Macintosh Common Lisp supports multiple processes. This allows multiple operations to occur simultaneously, including compilation, editing, and other event handling.

You can abort a process (except a small number that cannot be interrupted) by pressing Command-period or by choosing Abort from the Lisp menu. You can suspend a process and enter a break loop to examine it by pressing Command-comma or by choosing Break from the Lisp menu.

There are often only two processes running: the main process, and a process which handles events. If there are only two processes running, then Command-period and Command-comma apply to the main process. To abort or break the event process, use Option-Command plus period or comma. If there are more than two processes running, a dialog appears allowing you to choose a process to abort.

Choose Continue or press Command-slash to resume the process that has been suspended. Continue is also available on the list generated by the Restarts command on the Lisp menu.

Certain MCL tasks (for example, garbage collection) cannot be interrupted. During these operations no other tasks can be performed and no other processes will be given time.

The stepper

Macintosh Common Lisp provides the `step` macro as a simple way of stepping through the execution of an expression, subexpression by subexpression.

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 a true value. 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 options to the `trace` macro.

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

Figure 4-7   Stepping through the `factorial` function
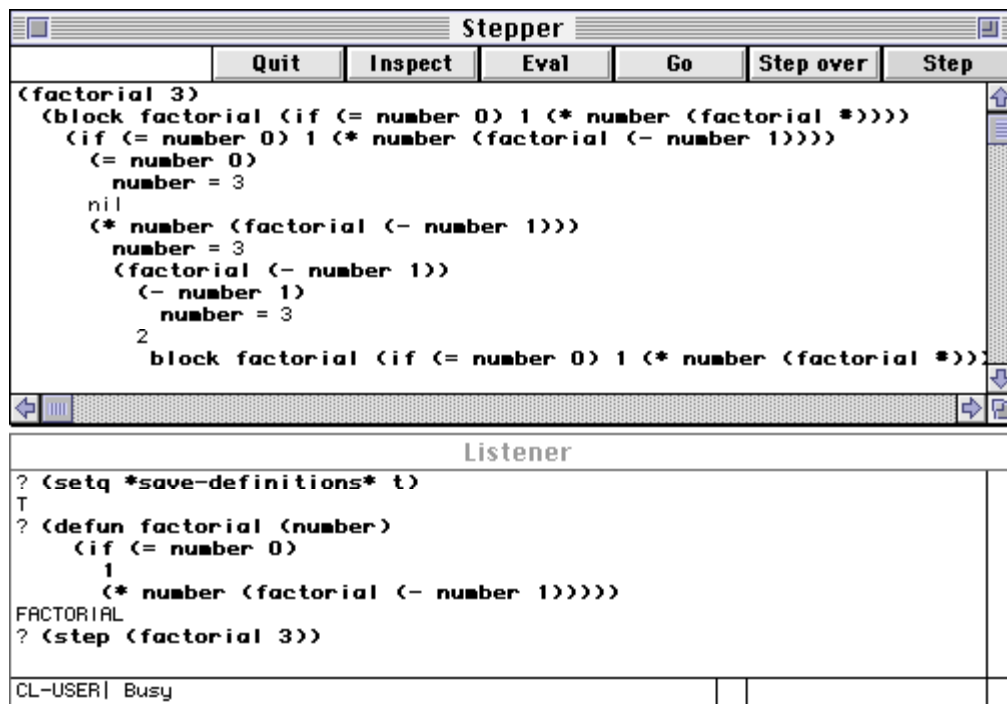
Trace

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.

Note that self-recursive function calls are normally compiled inline. In order to be able to trace such calls, the function must be declared not-inline.

```
? (defun fact (num)
    (declare (notinline 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 Trace window is accessible through the Trace… command on the Tools menu. It shows you which functions are currently being traced, and lets you select functions to trace with a number of options. For example, it lets you easily choose methods to trace within a generic function.

What you've learned about debugging

Macintosh Common Lisp has a variety for tools to help you understand and debug your program. There are tools for retrieving documentation and other information about definitions. There are tools for inspecting objects in a variety of ways. Finally, there are tools for working with and recovering from error situations.

Chapter 5:

Sources of Additional Information

Contents

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

Common Lisp References

Two Common Lisp reference works are available. These are not tutorials, but are complete descriptions of every feature of the language.

■ *Common Lisp: the Language, second edition* is available in HTML format in the documentation folder of the MCL CD.

■ The ANSI Common Lisp standard (X3.226-1994) is available in HTML format on the internet, at
`<http://www.harlequin.com/books/HyperSpec/>`

MCL implements the language as described in *Common Lisp: the Language, second edition*. However, this version is close enough to the ANSI standard that the latter is still quite a useful reference when using MCL.

Common Lisp Tutorials

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.

Graham, Paul. *ANSI Common Lisp.* New York: Prentice Hall, 1995.

Koschmann, Timothy. *The Common Lisp Companion.* Englewood Cliffs, NJ: 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.

Tanimoto, Steven. *The Elements of Artificial Intelligence Using Common Lisp.* Computer Science Press, WH Freemand and Company, 1990.

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 an excellent introduction to programming, abstraction, and computer science. It uses Scheme, a language similar to Common 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

Appendix B: The Common Lisp Object System provides a short introduction to CLOS.

The following book by Sonya Keene is a very thorough tutorial on CLOS (the Common Lisp Object System).

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.

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. 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 the Apple Developer Catalog (formerly APDA). *Inside Macintosh* is available in paper and CD editions.

Examples

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.

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. This appendix just describes the starting point.

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 the Emacs tradition, these modifiers are called Control and Meta. The following key sequences are used to indicate Control and Meta in MCL:

■ To enter a Control 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 Window), hold down the Control key and press X, then continue to hold down Control and press S. To enter Control-X H (Select All), hold down Control and press X, then release the Control key and press H.

■ There are two ways to enter a Meta keystroke.

By default, the Option key used to indicate Meta keystrokes. To enter a Meta keystroke, hold down the option key and then press the other key of the keystroke.

The example file "escape-key.lisp" allows the escape key to be used as the Meta keystroke. When this file is loaded, you enter a Meta keystroke by pressing and releasing the escape key, and then pressing and release the other key of the keystroke. This allows you to have access to the optional character set of the Macintosh when editing in Fred.

■ To enter a Control-Meta keystroke, hold down both modifier keys 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.

■ **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.

■ **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.

■ **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 keystroke sequences shown in Table A-4 are used to insert text and space.

■ **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

■ **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 keystroke sequences shown in Table A-5 are used to delete text and spaces.

■ **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 execute, compile, macroexpand, and read the current Lisp expression.

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

Function	Keystroke
Evaluates or compile current expression	Enter
Evaluates or compiles current selection or current top-level expression	Control-X Control-C
Evaluates current expression	Control-X Control-E
Repeatedly macroexpands the current expression until the result is no longer a macro. The result of each macroexpansion is pretty-printed to the Listener.	Control-M
Repeatedly macroexpands the current expression until the result is no longer a macro. The final expression is pretty-printed to the Listener, but the intermediate expressions are not.	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 windows, open files, and select windows.

■ **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
Selects the second window. Pressing this keystroke repeatedly swaps the top two windows.	Control-Option-L

Incremental search

Fred supports keyboard-based incremental searching.

You begin an incremental search by pressing Control-S (search forwards) or Control-R (search reverse). After typing this command keystroke, you begin typing the string for which you are searching. As each character is typed, the search proceeds to the next occurrence in the window of the string as specified so far. You can search for the next occurrence without augmenting the string by typing an additional Control-S or Control-R.

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.

■ **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:

The Common Lisp Object System

Contents

This appendix provides a simple introduction to CLOS, the Common Lisp Object System.

MCL and CLOS

Macintosh Common Lisp uses the **Common Lisp Object System (CLOS),** the object system of ANSI Common Lisp. CLOS is based on many years of experience with Common Lisp, augmented by substantive proposals and changes suggested by its developers and users, and coordinated by X3J13, a subcommittee of ANSI committee X3.

Common Lisp: The Language, second edition provides detailed summaries of the X3J13 committee's thinking. Though not an official standards document, the second edition of Steele extensively documents almost all of the functionality contained in the standard. Some changes to the standard have been approved since the time of publication of *Common Lisp: The Language*. Where they affect Macintosh Common Lisp, these changes are documented in the appropriate sections of this manual.

As you write new code, you should consult the standard as described in Steele or in the ANSI specification.

Definitions

The basic concepts of CLOS include classes, slots, instances, generic functions, and methods.

Classes and their superclasses

A **class** is an object that determines the inherited behavior of other objects, called its **instances.**

■ Classes are organized in a hierarchy.

■ A class can inherit structure from other classes, which are called its **superclasses.** Its immediate superclasses are called its **direct superclasses.** A class is a **subclass** of each of the classes from which it inherits.

- There are two classes at the top of the class hierarchy. The class named `t` has no superclasses and is a superclass of every class except itself. The class named `standard-object` is a superclass of all the classes programmers create. (However, it is not a superclass of some built-in classes.)

- A class has a name.

- A class has a **class precedence list,** which is the total ordering of the class and all its superclasses. When a class is defined, the order in which its direct superclasses are mentioned determines the order of the class precedence list.

Most Common Lisp types now correspond to a class with the same name. That is, for a type `macptr`, there will also be a class `macptr`.

Slots

A class is associated with a set of **slots**.

Slots are used for storing information associated with a class and its instances. A slot is described by a **slot specifier.** Each slot specifier includes the name of the slot and zero or more slot options. A class is also associated with a set of **class options,** which are slot options with a common value for the whole class. Slot options and class options control the following:

- determining default initial values for a given slot

- reading and writing the values of slots

- controlling whether a value for a given slot is shared by all instances of a class or whether each instance has its own value for the slot

- supplying a set of initialization arguments and default values to be used when instances are created

---

Instances

In CLOS, there is a clear distinction between classes and instances. CLOS is not a prototype-based object system.

- An **instance** is a Common Lisp object, one of a group of zero or more instances of a class. An instance may have a default state derived from the class definition, or may have its own particular state.

- An instance may use a method defined on its class or on one of the class's superclasses, or may have its own method. See the next section for the definition of methods.

- Every Common Lisp object, including class objects, is an instance of some class.

Generic functions and methods

Behavior is expressed through **generic functions,** Lisp functions whose behavior depends on the classes or identities of the arguments supplied.

- A generic function has **methods** for a number of classes. That is, the generic function has a number of ways to perform a procedure, each way designed to be used for a specific class. For example, the generic function `view-draw-contents` has one method for simple views, one for views, one for windows, and so on. That is, it knows how to draw the contents of a simple view, the contents of a view, and so on.

- These methods are related but usually not identical (for instance, the `view-draw-contents` method for `simple-view` doesn't need to handle subviews, but the one for `view` does).

- When you want to perform an operation on some objects, you apply a generic function to an instance. By determining which methods of the generic function are associated with the instance's class, the generic function knows which of its methods to apply to the instance.

- Methods are applied to instances through method combination. A generic function determines one **primary method** suitable for an instance—one basic procedure that is the most appropriate to the instance. It may also call other methods through `call-next-method`. One or more **auxiliary methods** may be run before or after the primary method. The entire group of methods applicable to the instance is called the **effective method.**

- Methods have arguments, just as functions do.

- Methods of the same generic function must always have the same number of required and optional arguments; that is, they must have **congruent lambda lists.**

- The required arguments of a method are **specialized;** that is, each argument is associated with a class or an instance. The specializers of a method determine whether the method is appropriate for a given set of arguments.

Classes and instances

Classes can be created and redefined. You can create instances of classes and give values to slots in a class or an instance.

Creating a class with the macro `defclass`

The macro `defclass` creates a new class. Its first argument is a list of the superclasses of the new class. If the argument is `nil`, the new class is based on `standard-object`.

Its second argument is a list of slot specifiers. Each slot is a list, the first element of which is a symbol that names the slot. The second and third elements are the slot's **initialization argument** and default **initial value form,** if it has them. These appear in definitions as the `initarg` and `initform` keywords.

Initialization arguments are keyword arguments used to supply the values for slots when new instances are created. Initial value forms provide a mechanism for a user to give a default initial value form for a slot.
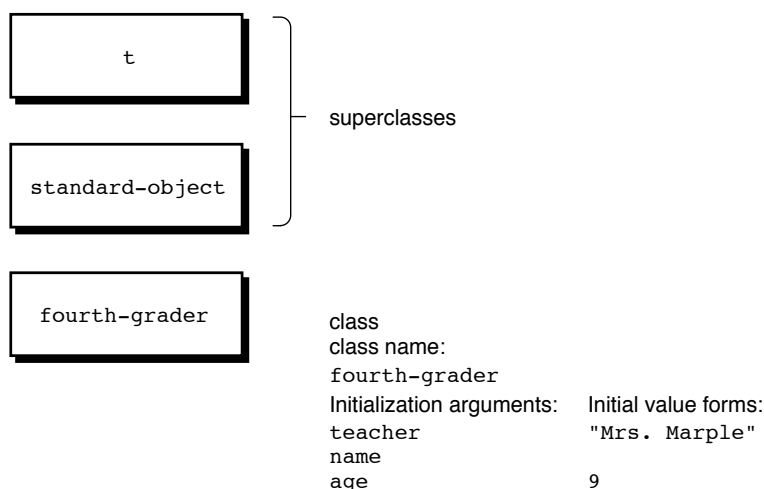
 The macro `defclass` returns the new class object.

Here is an example of the use of `defclass`.

```
? (defclass fourth-grader ()
    ((teacher :initarg :teacher
              :initform "Mrs. Marple")
     (name :initarg :name)
     (age :initarg :age :initform 9)))
#<STANDARD-CLASS FOURTH-GRADER>
```

Figure C-1 shows a graphic representation of how the class `fourth-grader` is built. This class is associated with three slots—`name`, `age`, which has the default initial value 9, and `teacher`, which has the default initial value `"Mrs. Marple".` The name of the new class is the symbol `fourth-grader`.

Figure C-1   The class `fourth-grader`


```
                                        class
    fourth-grader                       class name:
                                        fourth-grader
                                        Initialization arguments:   Initial value forms:
                                        teacher                     "Mrs. Marple"
                                        name
                                        age                         9
```

Creating an instance and giving its slots values

To create an instance of the class, use the generic function `make-instance`. This function creates and returns a new object based on its argument, which should be a class.

```
? (setq john (make-instance 'fourth-grader))
#<FOURTH-GRADER #x436B51>
```

An instance has slots as determined by its class and that class's superclasses.

You can set the values of an instance's slots when you create it. The following example creates an instance of `fourth-grader` and uses the initialization argument `:teacher` to override the default value for that slot:

```
? (setf john (make-instance 'fourth-grader
                                 :teacher "Ms. Hsu"))
#<FOURTH-GRADER #x477181>
```

The function `slot-value` retrieves the value of a slot. This function takes two arguments, the class or instance and the name of the slot.

```
? (slot-value john 'teacher)
"Ms. Hsu"
```

You can set the value of most slots of an already created instance by using `setf` with the name of the slot, for example, to give the instance `john` its own name (Figure C-2) or to change the value of the `:teacher` slot.
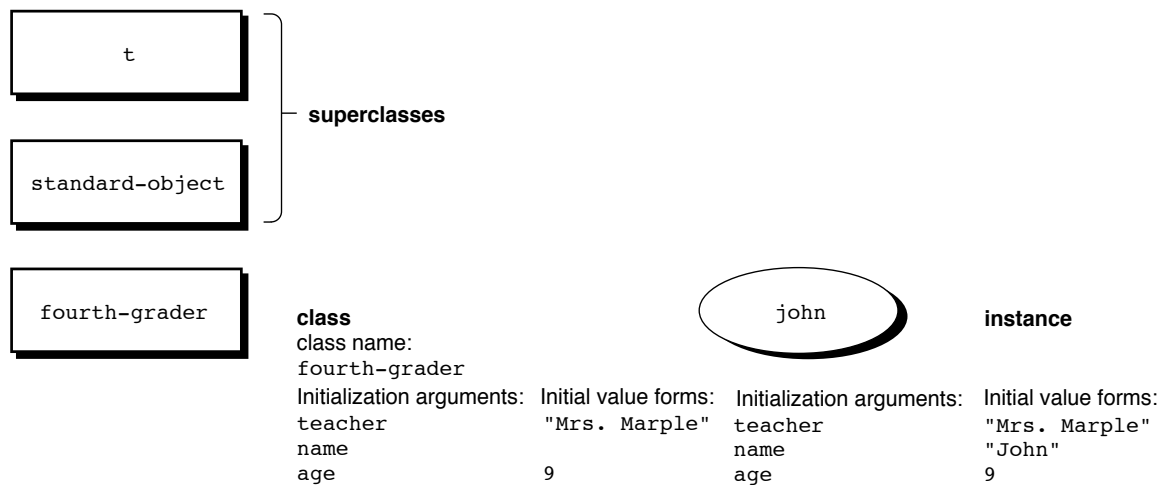
```
? (setf (slot-value john 'name) "John")
"John"
? (slot-value john 'name)
"John"
? (setf (slot-value john 'teacher) "Ms. Miller")
"Ms. Miller"
? (slot-value john 'teacher)
"Ms. Miller"
```

◆   *Note:* Accessor methods provide a simpler syntax than `slot-value`
    does. For more details, see "Creating and using accessor methods" on
    page 94.

Figure C-2   An instance of `fourth-grader` with a value in the `name` slot

To find the slot value associated with an instance, CLOS first looks up
the value associated with the slot at the instance level. If the slot is
unbound at the instance level, CLOS looks up the value associated with
the instance's class. If the slot is still unbound, CLOS looks for the value
in the slot associated with the first class that is a member of the class's
class precedence list, then looks in the slot of the second class, and so on
until it finds one value, which it returns.

Redefining a class

To redefine a class, simply edit the previous definition of the class and execute it again. Here is a revised definition of fourth-grader, adding a school slot :

```
? (defclass fourth-grader ()
    ((teacher :initarg :teacher
               :initform "Mrs. Marple")
     (school :initarg :school
               :initform "Lawrence School")
     (name :initarg :name)
     (age :initarg :age :initform 9)))
#<STANDARD-CLASS FOURTH-GRADER>
```

After the redefinition, all instances of fourth-grader, including ones already created, will have a school slot.

```
? (slot-value john 'school)
"Lawrence School"
```

When you create a new instance, you can specify a value for the school slot.

Allocating the value of a slot in a class

You can specifically instruct a slot to be associated not with an instance, but with the instance's class, by using the :allocation option when you define the slot.

The two :allocation options are :instance and :class. The :instance option is the default; this option means that each instance of a class has local storage for the slot that can be associated with its own value. The :class option specifies that the class stores the value of the slot. A slot of this kind is called a **shared slot** or a class slot. The following code defines picky-fourth-grader, associating it with a class slot for subjects-studied:

```
? (defclass picky-fourth-grader
    (fourth-grader)
    ((subjects-studied
      :initarg subjects-studied
      :initform '(English math woodworking)
      :allocation :class)))
#<STANDARD-CLASS PICKY-FOURTH-GRADER>
```

Since all instances of `picky-fourth-grader` access the same `subjects-studied` slot, you can `setf` the class slot value through any instance:

```
? (setf zane (make-instance 'picky-fourth-grader))
#<PICKY-FOURTH-GRADER #x40CDC9>
? (setf justus (make-instance 'picky-fourth-grader))
#<PICKY-FOURTH-GRADER #x40EFC8>
? (setf (slot-value zane 'subjects-studied)
    '(English math woodworking phenomenology))
(ENGLISH MATH WOODWORKING PHENOMENOLOGY)
? (slot-value justus 'subjects-studied)
(ENGLISH MATH WOODWORKING PHENOMENOLOGY)
```

Classes as prototypes of other classes

As you can see from the example "Creating an instance and giving its slots values" on page 88, you can build classes from other classes. Subclasses may add slots or change the default initial value of a slot that already exists. Subclasses may also have their own methods (see the next section, "Methods").

This is the syntax for defining a new class. It defines `modern-fourth-grader`, built on `fourth-grader`, with an additional slot, `computer`, and a new default value for `teacher`.

```
? (defclass modern-fourth-grader
    (fourth-grader)
    ((computer :initarg :computer
               :initform "Macintosh")
     (teacher  :initform "We use HyperCard stacks.")))
#<STANDARD-CLASS MODERN-FOURTH-GRADER>
? (setf mariah (make-instance 'modern-fourth-grader))
#<MODERN-FOURTH-GRADER #x51A209>
? (slot-value mariah 'teacher)
"We use HyperCard stacks."
? (slot-value mariah 'computer)
"Macintosh"
? (setf (slot-value mariah 'teacher)
        "We program our own in Macintosh Common Lisp.")
"We program our own in Macintosh Common Lisp."
```

The instance `mariah` inherits slots and values it does not redefine.

```
? (slot-value mariah 'school)
"Lawrence School"
```

Methods

CLOS is built on the idea that many functions operate in different ways when called on different classes of objects. Any function that operates in more than one way is called a **generic function,** a Lisp function whose behavior depends on the parameters supplied to it. Like any other Lisp function, a generic function can be passed as an argument and returned as a value, and it does all the other things a function does.

But while an ordinary function has a single body of code that is always executed, a generic function has a set of bodies of code, called **methods.** When a generic function is called on an instance, the function looks at the class or classes of the instance. In the simplest case, the generic function has a method for the instance's class. (For example, the generic function `#'set-part-color` has a method for `table-dialog-item`. Calling `#'set-part-color` on an instance of `table-dialog-item` calls that method. Calling `#'set-part-color` on an instance of `radio-button-dialog-item` calls a different method.)

In cases where the instance's class has a complex parentage, methods may be combined. What the function does—which bodies of code are called and how they are combined—is calculated from the **class precedence list,** the total ordering of the set of classes from which a class inherits. It also depends on the method combination type.

You can define a generic function by using the macro `defgeneric`. More often, however, you use `defmethod`, which defines a method on a generic function for a particular class. If the generic function does not already exist, `defmethod` creates one.

Defining a method and creating a generic function

By using `defmethod` to define a method, you automatically generate a generic function with the same name as the method. For example, you can create a generic function `#'say` by using `defmethod` to define a method `say`. Here is an example of the use of `defmethod`.

The following function prints a sentence of an instance of `fourth-grader` surrounded by enthusiastic punctuation, then returns `nil`:

```
? (defmethod say ((child fourth-grader) sentence)
    (princ "***")
    (princ sentence)
    (princ "!***")
    (terpri))
SAY
? (setq billy-crystal (make-instance 'fourth-grader
                          :name "Billy Crystal"))
#<FOURTH-GRADER #x60B5F1>
```

When `say` is called on `billy-crystal`, it checks the type of `billy-crystal` and finds that it is a `fourth-grader`. The generic function checks to see whether it has a method for `fourth-grader`; since it does, it runs that method.

```
? (say billy-crystal "Marvelous")
***Marvelous!***
NIL
```

Congruent lambda lists

All methods on a generic function have congruent lambda lists. That is, they have the same number of required and optional arguments. (For full details on congruent lambda lists, see *Common Lisp: The Language*, pages 791–792.) For example, the generic function `#'say` always requires an instance to which it is applied and a sentence.

Defining methods on instances

You can specialize methods on individual instances as well as on entire classes. If you need a method for `say` that is called only on `alisa`, an instance of `fourth-grader`, then specify `child` to be `eql` to `alisa`, as shown in the following example. The expression `(declare (ignore sentence))` avoids a compiler warning. The function `call-next-method` calls the next most specific method, which is the method on `fourth-grader`:

```
? (defmethod say ((child (eql alisa)) sentence)
   (declare (ignore sentence))
   (call-next-method)
   (princ "I have an HP calculator")
   (terpri))
#<Method SAY ((EQL #<FOURTH-GRADER #x436B51> T)>

? (say alisa "Hello")
***Hello!***
I have an HP calculator
NIL
```

Applied to other fourth-graders, such as sharon, say still calls its usual
fourth-grader method.

```
? (say sharon "Hello")
***Hello!***
NIL
```

◆   *Note:* The function `call-next-method` is perfectly acceptable here,
    but CLOS more commonly would use an `:after` method. Method
    combination is discussed later in "Creating auxiliary methods and
    using method qualifiers" on page 100.

Creating and using accessor methods

The macro `defclass` provides syntax for automatically generating
methods to read and write slots. Because the accessor is the most
common of these methods, the whole group is called **accessor methods.**
You can request three kinds of methods.

■   If you request a *reader*, Macintosh Common Lisp generates a method
    for reading the value of a slot but none for storing a value into it.
    Readers are used when the slot value won't change.

■   If you request a *writer*, Macintosh Common Lisp generates a method
    for storing a value into a slot, but no method for reading its value.

■   If you request an *accessor*, Macintosh Common Lisp generates two
    methods, a named method for reading the value and a `setf` method
    on the named method for storing a new value.

Using accessors to read and write the values of slots is preferable
because accessor methods hide implementation detail. Your clients can
call the accessor method without knowing whether it accesses a slot or
computes a value in some other way. The implementation can later
change without affecting the client's code.

Here is the `fourth-grader` class redefined using accessor methods. The accessor doesn't need to have the same name as the slot; here the accessor of the slot `teacher` is named `fourth-grade-teacher`:

```
? (defclass fourth-grader ()
    ((teacher :initarg :teacher
              :initform "Mrs. Marple"
              :accessor fourth-grade-teacher)
     (name :initarg :name
           :reader name)
     (school :initarg :school
             :initform "Lawrence School"
             :accessor school)
     (age :initarg :age
          :initform 9
          :accessor age)))
```

The `:reader` and `:accessor` methods simply provide a more abstracted alternative to `slot-value` and do not prevent your using `slot-value` as well. Creating the accessor method `fourth-grade-teacher` is equivalent to

```
? (defmethod fourth-grade-teacher ((student fourth-grader))
    (slot-value student 'teacher))


? (defmethod (setf fourth-grade-teacher)
             (new-teacher (student fourth-grader))
    (setf (slot-value student 'teacher ) new-teacher))
```

Making an instance of `fourth-grader` works the same way as previously.

```
? (setq delia (make-instance 'fourth-grader
                                :teacher "Mr. Smith"
                                :name "Delia"))
```

But you can now use the accessor method `fourth-grade-teacher` to get the `teacher` of `delia` and the accessor method `age` to get the age of `delia`:

```
? (fourth-grade-teacher delia)
"Mr. Smith"
? (age delia)
9
```

Accessor methods create regular generic functions and can be used just like any other Lisp functions. For instance, you can combine `fourth-grader` methods `say` and `school` to have an instance of `fourth-grader` report on the school it goes to. Because all look-ups occur at run time, this works even though `say` was defined before `fourth-grader` was redefined.

```
? (say mariah (school mariah))
***Lawrence School!***
NIL
? (say alisa (fourth-grade-teacher alisa))
***Mrs. Marple!***
I have an HP calculator
NIL
? (setf (age sharon) 10)
10
```

Customizing initialization with `initialize-instance`

Creating instances is not only a matter of associating values with slots;
other initialization must often be performed. You specify what needs to
be done by specializing the generic function `initialize-instance`.
A method on `initialize-instance` for `fourth-grader` might
look like the following:

```
(defmethod initialize-instance ((child fourth-grader)
                                    &rest initargs)
   (add-to-class-list child)
   (check-vaccinations child)
   (check-special-programs child))
```

When an instance of `fourth-grader` is created, `make-instance`
calls `initialize-instance`, which calls the functions `add-to-
class-list` and so on. These functions run their methods for
`fourth-grader`. The functions that `make-instance` calls do not
need to be generic functions and, if they are, do not need to have
methods specifically for `fourth-grader`; they might have methods
for one of the class's superclasses.

Creating subclasses and specializing their methods

When you create a subclass, you can write methods specific to that
subclass.

```
? (defclass shy-kid (fourth-grader)())
#<STANDARD-CLASS SHY-KID>
? (defmethod say ((child shy-kid) sentence)
   (princ "...")
   (princ sentence)
   (princ "...")
   (terpri))
#<Method SAY (SHY-KID T)>
```

```
? (setq max (make-instance 'shy-kid))
#<SHY-KID #x609A69>
? (say max "Hi")
...Hi...
NIL
```

An instance of a subclass still inherits slot values and methods from its class's superclasses.

```
? (school max)
"Lawrence School"
```

Method combination

◆ *Note:* This and the following sections provide a very simplified summary of method combination. For full details see *Common Lisp: The Language*.

How does a function decide which method to use for a particular set of arguments?

There are two categories of methods. The first, the **primary method,** defines the main action of the effective method that the generic function applies to the instance.

The second category, **auxiliary methods,** may modify that action in one of three ways. An auxiliary method has the method qualifier `:before`, `:after`, or `:around`. They run before the primary method, after the primary method, or around the primary method and all of its `:before` and `:after` methods.

The primary method

The primary method applied to the instance is always the most specific method the generic function has for the arguments it has been given. When a generic function has a method for an instance, the instance's method is used. If not, CLOS looks up applicable methods for the instance's direct superclass, then for that superclass's superclass, and so on. As soon as one applicable method is found, the search stops and that method is applied to the instance. (If there is no applicable primary method, the function signals an error.)

The primary method and the class precedence list

It is important to know in what order to look up methods. When a class has many superclasses, behavior is determined by the ordered list of its superclasses—its class precedence list. The class precedence list is basic to method combination.

The order in which the superclasses are listed determines the order in which their primary methods are consulted. If the instance has a method, that method is used. If the class has a method, that is used. Otherwise the look-up consults the first (leftmost) superclass in the class precedence list, looking up the class's methods and those of the class's superclasses, leftmost superclass first. If there is no appropriate method anywhere in the first superclass or its superclasses, the look-up proceeds to the second superclass, and so on. The first applicable primary method that is found is used.

However, when elements of the class precedence list have superclasses in common, CLOS examines, in this order:

1. all of the elements of the first superclass except the class in common

2. all of the elements of the second superclass except the class in common

3. the class in common (followed by its superclasses)

Examples of classes with multiple superclasses

We have seen simple class precedence at work in earlier examples. Let us look at some examples of multiple superclasses.

For example, suppose two classes, `bored-kid` and `happy-kid`, both subclasses of `fourth-grader`, with instances `harvey` and `laura`. The bored kid has homework and the happy kid has a pet. Each subclass has a method for a generic function, `#'go-crazy`:

```
? (defclass bored-kid (fourth-grader)
     ((homework :initarg :homework
                 :initform "Easy Steps to Calculus"
                 :accessor homework)))
#<STANDARD-CLASS BORED-KID>

? (defmethod go-crazy ((child bored-kid) exclamation)
     (princ "I'm so bored!  ")
     (princ exclamation)
     (princ "!  What a dump!")
     (terpri))
#<Method GO-CRAZY (BORED-KID T)>
```

```
?  (setq harvey (make-instance 'bored-kid))
#<BORED-KID #x608879>
? (go-crazy harvey "Yukh")
I'm so bored!  Yukh!  What a dump!
NIL
? (defclass happy-kid (fourth-grader)
    ((pet :initarg :pet
          :initform "gorilla"
          :accessor pet)))
#<STANDARD-CLASS HAPPY-KID>
? (defmethod go-crazy ((child happy-kid) exclamation)
    (princ "How happy I am!  ")
    (princ exclamation)
    (princ "!  I'm going to shout and throw paper clips!")
    (terpri))
#<Method GO-CRAZY (HAPPY-KID T)>
? (setq laura (make-instance 'happy-kid))
#<HAPPY-KID #x608C51>
? (go-crazy laura "Wow")
How happy I am!  Wow!  I'm going to shout and throw paper
clips!
NIL
```

When there is a conflict: Choosing between methods

When two classes inherit from the same superclasses, CLOS uses the
class precedence list to determine which primary method to use. For
example, we can define two new classes, happy-kid-with-
homework and bored-kid-with-pet, that inherit from the same
superclasses, but in opposite order.

```
? (defclass happy-kid-with-homework
    (happy-kid bored-kid)())
#<STANDARD-CLASS HAPPY-KID-WITH-HOMEWORK>
? (setq smiley (make-instance 'happy-kid-with-homework))
#<HAPPY-KID-WITH-HOMEWORK #x609C48>

?  (defclass bored-kid-with-pet (bored-kid happy-kid)())
#<STANDARD-CLASS BORED-KID-WITH-PET>
? (setq bad-max (make-instance 'bored-kid-with-pet))
#<BORED-KID-WITH-PET #x609C53>
```

To find the applicable method for go-crazy, CLOS looks for a method
associated with the instance, then with the class, then with the first
direct superclass in the list of the class's parents. The function go-
crazy has an applicable method for the first parent, happy-kid.
Therefore happy-kid-with-homework calls the go-crazy method
associated with happy-kid as a primary method. It never calls the
method for bored-kid because it doesn't get that far.

```
? (go-crazy smiley "Wow")
How happy I am!  Wow!  I'm going to shout and throw paper
clips!
NIL
```

But bored-kid-with-pet calls the bored-kid method.

```
? (go-crazy bad-max "Ugh")
I'm so bored! Ugh! What a dump!
NIL
```

Choosing between methods associated with direct and with more distant superclasses

Suppose that happy-kid has a method for #'say but bored-kid
doesn't:

```
? (defmethod say ((child happy-kid) sentence)
      (declare (ignore sentence))
      (princ "This primary method comes from happy-kid.")
    (terpri))
#<Method SAY (HAPPY-KID T)>
```

When CLOS looks for a method, it examines a class before its direct
superclasses and a direct superclass before all other direct superclasses
specified to its right in the list of the class's parents. Remember, if two
of the direct superclasses of a class have a superclass in common, then
the order is

1. all of the elements of the first superclass except the class in
    common

2. all of the elements of the second superclass except the class in
    common

3. the class in common (followed by its superclasses)

Within those constraints, the local ordering of the class precedence list
is determined by taking all the elements of the class and doing a
topological sort on them.

Here, bored-kid and happy-kid have a superclass in common,
fourth-grader, so bored-kid and happy-kid both affect the
behavior of bored-kid-with-pet before fourth-grader does:

```
? (say bad-max "I hate homework.")
This primary method comes from happy-kid.
NIL
```

Creating auxiliary methods and using method qualifiers

Only one primary method can be applied to any set of arguments, but
its behavior can be modified using auxiliary methods.

All `:before` methods are run before the primary method. CLOS looks for `:before` methods in the same order as it looks for primary methods: instance first, then class, then direct superclasses. This order is called *most specific first*. After all applicable `:before` methods have run, Macintosh Common Lisp calls the primary method, then the`:after` methods in reverse order, from the `:after` methods associated with `t` all the way down to the class and instance `:after` methods if there are any. (This order is called *least specific first*.)

All `:around` methods (which are seldom used) specify code that is to be called instead of other applicable methods, but can pass control to other methods, including the primary, `:before`, and `:after` methods. The effect is that the `:around` method runs "around" the other methods. An `:around` method uses `call-next-method` to pass control to the other methods.

```
Here is an example of :before and :after methods.
? (defmethod say :before ((child bored-kid) sentence)
    (declare (ignore sentence))
      (princ "First there is a :before method from bored-kid.")
    (terpri))
#<Method SAY :BEFORE (BORED-KID T)>

? (defmethod say :after ((child bored-kid) sentence)
    (declare (ignore sentence))
     (princ "Then there is an :after method from bored-kid.")
    (terpri))
#<Method SAY :AFTER (BORED-KID T)>

? (defmethod say :around ((child bored-kid) sentence)
    (declare (ignore sentence))
    (princ "This illustrates method combination.")
    (terpri)
    (call-next-method))
#<Method SAY :AFTER (BORED-KID T)>

? (say bad-max "I hate homework.")
This illustrates method combination.
First there is a :before method from bored-kid.
This primary method comes from happy-kid.
Then there is an :after method from bored-kid.
NIL
```

When `say` is called on the instance `bad-max`, CLOS looks first for `:around` methods. Then it searches for `:before` methods, starting with the most specific, and finds and runs a `:before` method associated with `bored-kid`. It runs the applicable primary method, which is associated with `happy-kid`. Then it looks for `:after` methods, starting with the least specific, and finds the one associated with `bored-kid`, which it runs.

There can be multiple `:before` and `:after` methods, as in the following example:

```
? (defmethod say :after ((child fourth-grader) sentence)
    (declare (ignore sentence))
     (format t "This is an :after method for fourth-grader."))
#<Method SAY :AFTER (FOURTH-GRADER T)>
? (defmethod say :after ((child happy-kid) sentence)
    (declare (ignore sentence))
    (princ "Everybody likes my ~A. \(This is an :after method
for happy-kid\)" (pet child))
    (terpri))
#<Method SAY :AFTER (HAPPY-KID T)>
? (setq yoichi (make-instance 'bored-kid-with-pet
                               :pet "dog Lizzie"))
#<BORED-KID-WITH-PET #x554FA9>
? (say yoichi "Nobody ever listens to me.")
This illustrates method combination.
First there is a :before method from bored-kid.
This primary method comes from happy-kid.
This is an :after method for fourth-grader.
Everybody likes my dog Lizzie. (This is an :after method for
happy-kid)
Then there is an :after method from bored-kid.
NIL
```

Combining all methods applicable to this instance produces the
effective method, the complete list of primary and secondary methods.

Mixin classes and auxiliary methods

Mixins are ordinary CLOS classes with useful values or methods that
"mix in" some special behavior. They are generally not used alone, but
add their specializations to another class. In Macintosh Common Lisp,
they usually appear first in the class precedence list. The class that adds
Fred behavior to a window or dialog item, `fred-mixin`, is a mixin.

Extended examples

The examples folder on the MCL CD includes many other examples of
the use of CLOS. See, for example, "shapes-code.lisp."

Appendix C:

The MCL Menubar

Contents

This appendix describes all the menus and menu commands in the MCL menubar.

Like all parts of Macintosh Common Lisp, the menubar is written in MCL and is fully extensible. You can add and rename menus, menu-items, and command-key equivalents. This appendix just describes the starting point.

The MCL Menubar

The Macintosh Common Lisp menubar contains six menus. The first three are familiar from other Macintosh applications:

- The Apple menu, indicated by an apple ( ), has two sections. The first has a single command which provides information about MCL. The second provides access to other Macintosh applications and files.

- The File menu is used to create and close editor windows, to load and compile files, to print, and to quit Macintosh Common Lisp.

- The Edit menu is used to edit and search through text.

The last three menus are specific to MCL:

- The Lisp menu provides tools for executing expressions and interrupting and continuing program execution.

- The Tools menu gives access to a variety of programming tools.

- The Windows menu lists all the windows open in MCL.

Balloon help is available for all MCL menu commands.

Apple menu

There is a single command on the Apple menu.

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

File menu

The File menu includes the following commands.⌘

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 Selection (⌘-D)

> If there's a selection in the top editor window, MCL attempts to parse this selection as a pathname and to create an editor window for the pathname's file.

New Listener Creates a new Listener window.

Close (⌘-W) Closes the current window. If the window is a Fred window that has been edited since it was last saved, MCL displays a dialog box asking you if you want to save the changes.

Save (⌘-S) Saves the contents of the active window to its associated file. If the window isn't associated with a file, the "Save As" command is invoked.

Save As… Allows you to specify a directory and filename, and saves the contents of the active window to that directory and 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 and filename.

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.

Load File… (⌘-Y)

> Allows you to select a file for loading into MCL. You may load both source code and fasl files.

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

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 menu

The Edit menu includes the following commands.

Undo (⌘-Z) Undoes the last editor command, if possible. The name of this command will change according to context, to show what will be undone.

Undo More Undoes previous editor commands in order.

Cut (⌘-X)   Deletes the selected region and places it in the Clipboard and on the kill ring.

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 copied onto the kill ring.)

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

Search… (⌘-F) Displays the String Search dialog box.

Search Again (⌘-G)
                    Repeats the previous search.

Font   Contains a submenu which allows the user to change the font of the insertion point or of the selected text. The current font of the insertion point or of the beginning of the selection is shown with a checkmark.

Font Size   Contains a submenu which allows the user to change the font size of the insertion point or of the selected text. The current font size of the insertion point or of the beginning of the selection is shown with a checkmark.

Font Style   Contains a submenu which allows the user to change the font style of the insertion point or of the selected text. The current font style of the insertion point or of the beginning of the selection is shown with checkmarks.

Font Color   Contains a submenu which allows the user to change the font color of the insertion point or of the selected text. The current font color of the insertion point or of the beginning of the selection is shown with a checkmark.

Word Wrap   Allows the user to enable and disable word wrap in Fred windows. By default, word wrap is desabled in new Fred windows. A check mark in the title of this command indicates that word wrap is enabled for the front Fred window.

Lisp menu

The Lisp menu contains the following commands:

Execute Selection (⌘-E)

Executes the current selection in the top editor window. If there is no selection and the insertion point is next to a parenthesis, the expression bounded by the parentheses is executed.

Execute All (⌘-H)

Executes the entire contents of the top editor window.

Abort (⌘-period)

Cancels the current computation and returns to the read-eval-print loop. If MCL was in a break loop, it leaves the loop.

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 the "Continue" command or by calling the function `continue`.

Continue (⌘-slash)

Continues the last operation halted by a break or by a continuable error.

Restarts… (⌘-backslash)

Provides a list of possible ways to restart the current operation.

Tools menu

The Tools menu contains the following commands:

Apropos   Displays the Apropos dialog box. This box accepts a string entered by the user and displays a table of symbols that contain the string.

Get Info…   Gets a symbol name from the user and displays various pieces of information about the symbol, allowing quick access to a number of tools for investigating the symbol.

Processes   Displays information about all current MCL processes.

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

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. MCL supports Common Lisp extended wildcards in pathnames. The search uses the Boyer-Moore algorithm, and is quite fast.

Trace   Displays a window showing functions which are traced, and allowing the user to trace additional functions.

Inspector   Contains several subcommands that support inspecting a number of aspects of the MCL environment.

Save Application…
                     Brings up a dialog box that allows the user to specify saving parameters and then save out the current Lisp image as a standalone application.

Extensions   Contains subcommands that load commonly desired extensions to MCL, individually or in groups. MCL users can place additional extensions on this menu, as described in the file "lib:mcl-extensions.lisp".

Extensions/Load Multiple…
                     Allows the programmer to load several extensions simultaneously, and then optionally save the resulting Lisp image to disk. Images are described in "Preserving programming sessions" on page 47, and in the MCL Reference Manual.

Extensions/Save Image…
                     Allows the programmer to save an image of the current Lisp session. Images are described in "Preserving programming sessions" on page 47, and in the MCL Reference Manual.

Backtrace (⌘-B)
                     Brings up the stack backtrace window.

Fred Commands
                     Displays a window listing all the commands available in Fred, the MCL editor.

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

Preferences…   Brings up a dialog box that allows the user to set the values of various parameters and global variables of the Lisp environment.

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.

Index

Symbols

#@, point syntax 43

A

Abort menu command 107
aborting from errors 60
About Macintosh Common Lisp menu
    command 104
accessor methods of slots 94–96
`:accessor` slot option 95
actions of dialog items 44
`add-menu-items` function 46
`add-subviews` function 44
`:after` methods 97, 101
`:allocation` slot option 90
announcements related to MCL 14
anonymous functions 44
ANSI Common Lisp standard 12, 70
    and CLOS 84
Apple menu 104
`apropos` function 53
Apropos menu command 107
`arglist` function 51
`*arglist-on-space*` variable 52
argument lists
    congruency of 86, 93
    displaying 39, 51–52
`:around` methods 97, 101
auto-indentation 34
auxiliary methods 86, 97, 100–102

B

Backtrace menu command 108
`*backtrace-on-break*` variable 64
balloon help 104
`beep` function 44
`:before` methods 97, 101
break loops 59–65
Break menu command 107
bug reports 15
`button-dialog-item` class 44

buttons 44

C

`call-next-method` function 94
canceling operations 60
cancellation from errors 60
class names 85
class options for slots 85
class precedence lists 85, 92, 98
`:class` slot option 90
class slots 90–91
classes 84–85, 87–91
    defining 87–88
    redefining 90
Clear menu command 106
CLOS
    tutorial for 84–102
    tutorials 71
Close menu command 105
closing all windows of a class 37
Common Lisp Object System
    See CLOS
Common Lisp references 70
Common Lisp: the Language, second edition
    12, 70
Common Lisp tutorials 70
comp.lang.lisp.mcl newsgroup 15
compilation, incremental 25
Compile File… menu command 105
compiled files 40
`compile-file` function 38
compiling files 38–40
congruent lambda lists 86, 93
Continue menu command 107
control keystrokes in Fred 74
control-g in the Listener 28
control-option-n in the Listener 28
control-option-p in the Listener 28
control-return in the Listener 28
Copy menu command 106
creating
    classes 87–88
    Fred windows 30
    generic functions 92–96
    instances 88–89
    methods 92–96
    subclasses 96
    windows 42, 45

customizing MCL 25, 46, 47
Cut menu command 106
cutting and pasting 34

D

`defclass` macro 87
`defmethod` macro 92
deletion commands in Fred 79
delimiters, matching 32
dialog items 43–46
    actions of 44
Digitool
    how to contact 16
    web page of 15
direct superclasses 84
discussions of MCL on the Internet 14
`documentation` function 52
documentation strings
    displaying 39
    for built-in and user definitions 52
documentation, online 50–52

E

Edit menu 104, 105–106
editable text dialog items 44
`editable-text-dialog-item` class 44
`edit-definition` function 50
editor
    See Fred
effective methods 86
Emacs 24, 28, 30
    cutting and pasting commands 34
enter keystroke, in the listener 27
error messages 59–60
errors 59–65
escape key, in Fred 74
evaluation 25, 26
    See also execution
examples 72, 102
Execute All menu command 107
Execute Selection menu command 107
executing expressions in a Fred window 31
exiting MCL 40
Extensions menu command 108

F

".fasl" file extension 38
fasl files 38
`*fasl-save-definitions*` variable 51
`*fasl-save-local-symbols*` variable 51,
        65
file compilation 38–40
file mapping 20
File menu 104, 104–105
floppy disk installation 22
Font Color menu command 106
Font menu command 106
Font Size menu command 106
Font Style menu command 106
fonts, setting in an editable text dialog item 45
`format` function 34, 45
Fred 30–36
    deletion commands 79
    help commands 75
    incremental search 81–82
    insertion commands 77–78
    Lisp operation commands 80
    modifier keys 74
    movement commands 76
    navigation commands 76
    selection commands 77
    window commands 80–81
Fred command help 35
Fred Commands menu command 108
Fred windows
    creating 30
    executing expressions in 31
    indentation in 34
free space 53
ftp site for MCL 15
functions
    See also methods, generic functions
functions, anonymous 44

G

generic functions 86, 92
    defining 92–96
Get Info… menu command 107
`get-string-from-user` function 32

H

help
  for Fred commands 35
  for Listener commands 28, 37
help commands in Fred 75
hiding windows 37

I

images 47–48
incremental search in Fred 81–82
info-mcl mailing list 14
init file 25
init.lisp 25
`initarg` keyword 87
`initform` keyword 87
initial value forms of slots 87
initialization arguments of slots 87
`initialize-instance` function 96
initializing instances 96
init-keywords for windows 43
inline functions and tracing 67
insertion commands in Fred 77–78
`inspect` function 55
Inspector 55–58
  modifying objects with 55
Inspector menu command 108
installation instructions 19–22
  for MCL 3.1 21
  for MCL 4.0 19
  for MCL 4.0 "Demo Version" 19
  from floppies 22
`:instance` slot option 90
instances 84, 87–91
  creating 88–89
  in CLOS 85–86
  initialization of 96
interface toolkit 47
Internet resources 14–16
introspection commands 53–58

K

keystrokes, help for 36
keyword arguments to `make-instance` 43
kill-ring 34

L

lambda list congruency 86, 93
`lambda` special form 44
launching MCL 24
  by double-clicking files 48
learning MCL 12–13
libraries
  required to restart saved images 48
library folder in the MCL folder 19
Lisp menu 104, 106–107
Lisp operation commands in Fred 80
lisp-based editing 32–35
List Definitions menu command 107
Listener 25–29
  saving contents of 28
Listener Commands menu command 108
Listener commands, help for 28, 37
Listener keystrokes 27–28
Load File… menu command 105
loading files 40
local variable names, preserving for debugging
    65
locating source code 40
locating symbols 53–55

M

Macintosh programming references 71
mailing lists relating to MCL 14
`make-instance` function 42, 88
matching delimeters 32
MCL 3.1 and 4.0 compared 11
memory available in a Lisp session 53
memory requirements
  of MCL 3.1 18, 21
  of MCL 4.0 18, 20
menu class 46
menubar
  built-in 104–109
  replacing 46
`menu-install` function 46
`menu-item` class 46
menu-items 46
menus
  built-in 104
  Edit menu 105–106
  File menu 104–105
  in the application framework 46–47
  Lisp menu 106–107