Allegro CL 3.0 for Windows printed documentation in PDF format

All the printed documentation for Allegro CL 3.0 for Windows is available in this *pdf* file and is readable with an Adobe Acrobat Reader. Please Note:

- There are links in each Table of Contents page. Clicking on a chapter or section (but not heading) title causes a jump to the associated page.
- There are no links in any index in this edition.
- **Many pictures do not appear**. An apparent incompatibility in the picture format results in their not appearing in the PDF file (you see blank space instead)
- If you choose 'Bookmarks and Page' from the View menu, you will see bookmarks for each chapter of each manual. Clicking on a bookmark will cause the associated page to be displayed (usually, the first page of the chapter).
- The pagination (<chapter>-<number>) used in the printed documents does not correspond to the pagination used in the *pdf* file (where all pages are numbered in order from the beginning). To find a page listed PT-2-2, go to the beginning of Chapter 2 in the *Programming Tools* manual (using the Acrobat Bookmarks). Go to page 2 of that chapter.

Copyright and Conditions of Use

Copyright 1996, Franz Inc. All Rights Reserved.

The PDF hypertext markup that implements the hypertext features of the Allegro CL 3.0.1 for Windows User Guide pages collectively the *User Guide*, is the property of Franz Inc.

Distribution of the User Guide as a hypertext PDF document on the Internet does not constitute consent to any use of the underlying hypertext PDF markup for redistribution of any kind, commercial or otherwise, either via the Internet or using some other form of distribution, in hypertext or otherwise.

Permission to copy, distribute, display, and transmit the User Guide is granted provided that copies are not made or distributed or displayed or transmitted for direct commercial

advantage, that notice is given that copying, distribution, display, and/or transmission is by permission of Franz Inc., and that any copy made is COMPLETE and UNMODIFIED, including this copyright notice and its date.

Permissions related to performance and to creation of derivative works are expressly NOT granted.

Permission to make partial copies is expressly NOT granted, EXCEPT that limited permission is granted to transmit and display a partial copy of the User Guide for the ordinary purpose of direct viewing by a human being in the usual manner that PDF browsers permit the viewing of such a complete document, provided that no recopying, redistribution, redisplay, or retransmission is made of any such partial copy.

Permission to make modified copies is expressly NOT granted.

Permission to add or replace any links or any graphical images to any of these pages is expressly NOT granted.

Permission to use any of the included graphical (GIF) images in any document other than the User Guide is expressly NOT granted.

Restricted Rights Legend

Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in (i) FAR 52.227-14 Alt III, (ii) FAR 52.227-19, (iii) DFARS 252.7013(c)(1)(ii), or (iv) the accompanying license Agreement, as applicable. For purposes of the FAR, the Software shall be deemed to be ``unpublished'' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Franz Inc., 1995 University Ave., Berkeley, CA 94704.'

Warranty disclaimer

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. IN NO EVENT WILL FRANZ INC. BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY INACCURACY OR ERROR IN THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

PREFACE

There are two volumes of bound documentation for Allegro CL for Windows. This is volume 1. Each volume contains several manuals. There is also a *Read This First* document, which not bound in with the rest of the documentation.

Here is a brief description of the documentation:

1. *Read This First*. This document is supplied loose. It contains information that was not put in the bound documentation.

Volume 1

2. *Getting Started*. This document describes how to install Allegro CL for Windows on your system and it gives information about running the product.
3. *Common Lisp Introduction*. This document is an introduction to the Common Lisp language and the Allegro CL for Windows implementation of it.
4. *Interface Builder*. The Interface Builder allows you to build an interface to your application interactively. It is described in this manual.
5. *Foreign Function Interface*. Allegro CL for Windows supports calling applications written in languages other than Lisp. This document describes the interface.

Volume 2

6. *Programming Tools*. This document describes the user interface to Allegro CL for Windows. In particular, it describes the Toploop window, the editor, the debugging facilities, etc.
7. *General Index*. An index to all documents in Volumes 1 and 2.

Professional version only

The Professional version of Allegro CL provides the facility to create standalone applications. User who purchase the Professional version also receive the following document:

Professional supplement. This document describes features available in the Professional version (but not in the standard version), including how to create standalone applications.

Each individual manual has a table of contents at the beginning and an index at the end. The *General Index* manual, in volume 3, is an index for all manuals in Volumes 1 and 2.

Allegro CL for Windows

Getting Started

version 3.0

October, 1995

Copyright and other notices:

This is revision 0 of this manual. This manual has Franz Inc. document number D-U-00-PC0-03-51017-3-0.

Copyright © 1992-1995 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademarks of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, Windows 95, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

Contents

1 Introduction

Welcome to Allegro CL for Windows. This is the introductory document which will get you started with the system and direct you to the other documents and help facilities which describe how to use it. This manual is divided into the following sections:

1. **Introduction**. The section you are now reading.
2. **Installation**. This section describes how to install Allegro CL for Windows.
3. **Documentation**. This sections describes the documentation for Allegro CL for Windows. This section contains a description of the notation used in Allegro CL for Windows documentation.
4. **When Allegro CL starts up**. This section briefly describes the features on the screen and refers you to more complete documentation.
5. **Features of the Microsoft Windows**. Allegro CL for Windows is designed to run under various versions of the Microsoft Windows. In this section we discuss which versions and we mention certain features of MS Windows that help you when running Allegro CL for Windows.
6. **Sources of information and patches**. Franz Inc. maintains a World Wide Web page from which a product FAQ, patches, and other useful things can obtained. Its URL and other sources of information are described in this section.
7. **Example files**. A number of example files are included with the Allegro CL for Windows distribution. These are described in this section.
8. **Anomalies and things to note**. This section describes behavior of Allegro CL for Windows which may be unexpected or unusual.
9. **Support**. This section briefly introduces product support. See the *Read This First* document for more information on product support.

You should have received a *Read This First* document with the Allegro CL for Windows distribution. Please be sure to scan this document before installing or using Allegro CL for Windows. It may contain important information which could not be included in the bound documents.

2 Installation

Allegro CL 3.0 for Windows will run under Windows 95, Windows NT 3.51, or Windows 3.1 or Windows for Workgroups with the Win32s upgrade supplied with the distribution.

The software is distributed on a CD. The CD contains both Allegro CL for Windows and Win32s.

Installing Win32s (Windows 3.1 or Windows for Workgroups users only)

If you are running Windows 95 or Windows NT, you do not need to install Win32s. If you are running Windows 3.1, you must install Win32s before installing Allegro CL for Windows. You may have a version of Win32s already installed on your machine. If it is earlier than the version supplied with the distribution, we strongly recommend upgrading because it includes many additional features (over earlier versions) which are used by Allegro CL for Windows. To install Win32s, do the following:

1. Switch to the Program Manager, making it the active program.
2. Put the CD in your CD Drive. We assume the CD drive is device D in this document. Replace D with the correct letter if necessary.
3. Choose **Run** from the Program Manager File Menu.
4. In the dialog that appears, enter **d:\win32s\setup.exe**
5. Answer dialog questions when necessary.

The Win32s installation includes the optional installation of FreeCell, a solitaire game that requires Win32s in order to run, and can thus be used as a check to ensure Win32s is installed properly.

If the system detects that the same or a later version of Win32s is installed, it will inform you and you should abort the installation of Win32s.

The *Read This First* document gives the exact version number of Win32s distributed with Allegro CL.

Allegro CL 3.0 for Windows installation

There are two versions of Allegro CL for Windows: the Professional version and the standard version. The box should say which version you have received. Both are installed the same way (the differences between the two versions is described at the end of this section). To install, do the following:

1. **Choose a directory** on a filesystem that has 15 Megabytes free. The directory should be empty. We assume you choose `c:\allegro`. Note that the directory name must follow the 8.3 name limitation (no more than 8 letters for the name, no more than 3 for the extension) even if you are running Windows 95 or Windows NT. (Users of those products will not again run into this limitation.)
2. **Insert the distribution CD** into the CD drive. We assume the CD drive is device D. Replace D with the correct letter if necessary.
3. **Run the program** `d:\allegro\setup.exe`. If you are running Windows 95, choose **Run** from the menu displayed by clicking on Start and enter `d:\allegro\setup.exe` in the dialog that appears. If you are using Windows 3.1, choose **Run** from the Program Manager File menu and enter `d:\allegro\setup.exe` in the dialog that appears.
4. **Note the serial number on the CD jewel case** (box). You are asked for this serial number, your name, and your organization soon after the installation process begins. Enter the serial number exactly as it appears on the jewel case.
5. **Enter the directory for installation** or accept the default `c:\allegro`. When you are asked where to install Allegro CL, you are offered the default location `c:\allegro`. Either click on OK to accept this location or enter a different location. The location must follow the 8.3 filename limitation even if that limitation does not otherwise apply to your machine. Whatever directory you choose must have 15 Megabytes of free space.

   Note: this is the same default as that specified for Allegro CL for Windows 2.0. If you have Allegro CL for Windows 2.0 already installed and wish both versions to be available, choose a directory other than the directory where Allegro CL 2.0 for Windows is installed.

Once the directory is specified, the installation runs to completion, after which a new Program folder (on Windows 95) or a new Program Group (for Windows 3.1) has been created. The folder or group is called **Allegro CL 3.0 for Windows**.

Three program items are installed by the Professional version (Allegro CL 3.0, Allegro CL Runtime, and Uninstall), two by the standard version (Allegro CL and Uninstall). We show the folders for the Professional and standard versions next. If you are running Windows 3.1, you will see a Program Group (icons in a box titled Allegro CL 3.0). The icons are similar to those in the Windows 95 folders. (Many Windows 95 users may close the program folder window and start Allegro CL from the Programs choice of the Start menu. It is not necessary to have the folder displayed.)

Here is the program folder for the Professional version:

And here is the program folder for the standard version:

You can now run Allegro CL for Windows. In Windows 95, either click on the Start button, selecting Programs, selecting Allegro CL 3.0 for Windows, and selecting Allegro CL or start the program from the program folder. In Windows 3.1, start Allegro CL by double-clicking on the Allegro CL icon in the Allegro CL 3.0 for Windows program group.

Reinstallation

You can re-install Allegro CL for Windows by following the same instructions. (It is not necessary to reinstall Win32s before reinstalling Allegro CL.) All files in the distribution will be overwritten by files on the disks with the same names. You may want to clear the directories where Allegro CL for Windows is stored (by using the Uninstall program, for example) but this is not necessary.

Uninstall Program

Following the Windows paradigm, an Uninstall program is included with Allegro CL 3.0 for Windows. If invoked, it will ask at least twice if you are sure about uninstalling. If you confirm that you are, all installed programs will be removed. The directories created as part of the installation will also be removed if they are empty (that is if the only files in them are associated with the distribution).

Filename length restriction (Windows 3.1 only)

Windows 3.1 has a 64 character limit on the length of pathnames of files. The longest path in the Allegro CL for Windows distribution is 33 characters long; for example:

```
c:\allegro\ext\comtab\emacsctb.lsp
```

uses 33 characters including *c:\allegro*. If you choose a directory name significantly longer that *c:\allegro*, so that the limit is exceeded for some files, those files will not be installed.

Professional and standard versions

There are two versions of Allegro CL for Windows: Professional and standard. The versions are incompatible: compiled Lisp (*fsl*) and image (*img*) files generated by one cannot be used by the other.

The Professional version contains everything the standard versions does plus:

- Support for interprocess communication (*fsl\socket.fsl*, *sockaux.dll*).
- Certain Common Control widgets (*fsl\grid*.fsl*).
- The runtime generator.
- Access to certain source code (for Common graphics, the Interface Builder, and the Grapher). Getting source code requires an additional license (but no additional charge for Professional version customers). You will find a Source Code License among the distribution material. Fill it out and return it along with the Registration Card and the sources will be sent to you on a diskette.

The extra products available to Professional version customers can be purchased individually by standard version customers. Please contact Franz Inc. for more information.

3 Documentation

Nine documents comprise the documentation set for Allegro CL for Windows. Several of these documents are bound together. They are:

1. *Read This First*. This document is supplied separate from the bound manuals. It contains information that was (because of printing schedules) not put in the bound documentation.

Volume 1

2. *Getting Started*. This is the document you are reading now. It describes how to install Allegro CL for Windows on your system and it gives information about running the product.
3. *Common Lisp Introduction*. This document is an introduction to the Common Lisp language.
4. *Interface Builder*. The Interface Builder can be used to build an interface to your application.
5. *Foreign Function Interface*. Allegro CL for Windows supports calling applications written in languages other than Lisp. This document describes the interface.

Volume 2

6. *Programming Tools*. This document describes the programmer interface and the functions and variables associated with the programming interface (the editor, debugger, inspector, etc.)
7. *General Index*. This document contains an index to the five printed and bound manuals listed above.

Professional supplement (Professional version only)

If you purchased the Professional version of Allegro CL for Windows, you will also have the following manual:

> *Professional* supplement. The manual describes features available in the Professional but not in the standard version of Allegro CL for Windows 3.0. Among these are the Runtime Generator, which allows you to create customized applications that run independently of Allegro CL for Windows; the Winsock interface; and Grid widgets.

The Online Manual

Descriptions of Common Lisp (including CLOS), Common Graphics, and many Allegro CL extensions can be found in the Online Manual. The Online Manual follows the usual WinHelp format. Bring up the manual by choosing **Manual Contents** from the Help menu.

The Online Manual is arranged into chapters, with the functionality associated with each chapter described within it. Choosing **Manual Contents** brings up the contents of the Online Manual, and you can use standard WinHelp tools to navigate about the document at that point. On the Contents page, you will find links to Common Lisp Contents, Common Graphics Contents, CLOS Contents, the Release Notes, a description of the Online Manual, and some other immediately useful entries.

You can produce a WinHelp word list for the Online Manual by clicking on **Search**, clicking on the **Find** tab in the dialog that appears, clicking on the **Next** button, and clicking on the **Finish** button. This takes a while the first time you do it (since an index file must be

created and stored) but it is worth doing because the additional search capability greatly enhances the usefulness of the Online Manual.

The **Manual Entry** choice allows you to go directly to a single entry. It brings up the documentation for the symbol nearest the text cursor. If there is no such symbol, or if the nearest symbol is not documented in the Online Manual, a message to that effect is printed in the status bar and the Online Manual does not come up.

Note that certain things, like format directives, are not named by symbols, so there is no way to go directly to their descriptions using **Manual Entry**. But because they are after the entry for **`format`**, you can go to that entry and then page down or search to find the specific description of interest, or you can the new WinHelp search facilities.

Besides the entries on the main contents, an entry which is likely to be of particular interest to many users is **Programming environment**. This entry (it is the last item in the Lisp Contents entry) provides information on garbage collection, and other details of the implementation.

The Online Manual is sometimes referred to as Online Help.

Other documents for starting out

All users should look at the *Programming Tools* manual since this document describes how to use Allegro CL for Windows. Users new to Lisp may wish to look at the *Common Lisp Introduction*. Experienced Lisp users need not consult that document.

The Interface Builder is a very useful tool for constructing Graphical User Interfaces to Allegro CL for Windows applications. The *Interface Builder* manual tells how to use it.

Pictures in the documentation

There are many pictures derived from screen shots in the documentation. In some cases, we have converted the color and grayscale images to black and white. On your machine, the corresponding images will be in color or grayscale. For the same reason, those images do not use 3D effects.

Most of the images were created on a Windows 95 machine. Users of Windows 3.1 will often see dialogs and other features quite different from what is illustrated. (Some images were created on a Windows 3.1 machine, so Windows 95 users will see something different.)

Note too that the contents of windows, menus, and dialogs often reflect details of your environment that are likely different from the environment on the machine where the pic-

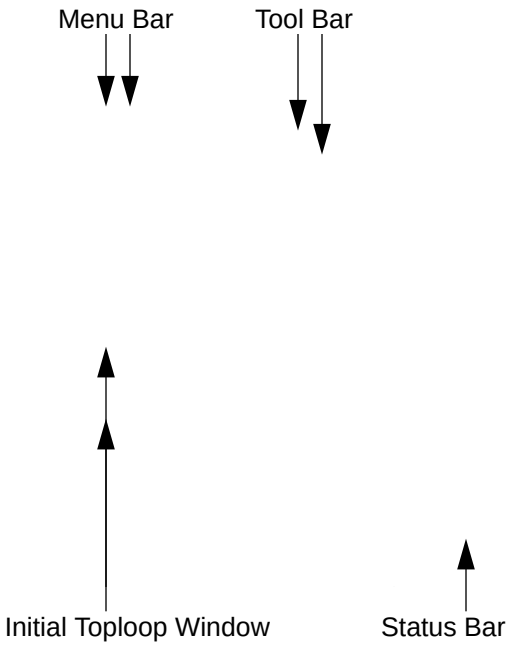
tures were made. Do not expect to see exactly what is illustrated. Instead, the pictures serve as a guide to what you will see.

Change bars

Change bars (a black line to the left of the text -- this paragraph has a change bar) are sometimes used to indicate new features or significantly changed functionality. The change is with respect to the 2.0 version of the manual. Where the functionality is the same, change bars are not typically used even if the text is rewritten.

4 When Allegro CL starts up

When you start Allegro CL for Windows, the screen will look something like this:



We have identified certain of the features:

- **The Menu Bar**. The standard Allegro CL menus are displayed. The contents of the menus are often self-explanatory. See section 2.5 **The menubar** in *Programming Tools* for more information.
- **The Toolbar**. Each button on the toolbar, when clicked on, either changes the state of Lisp or initiates some action. When the cursor is over a toolbar button, a description of what clicking on the button does is printed in the status bar. The toolbar can be hidden (and redisplayed) with the F12 key. See section 2.6 **The toolbar** in *Programming Tools* for more information.

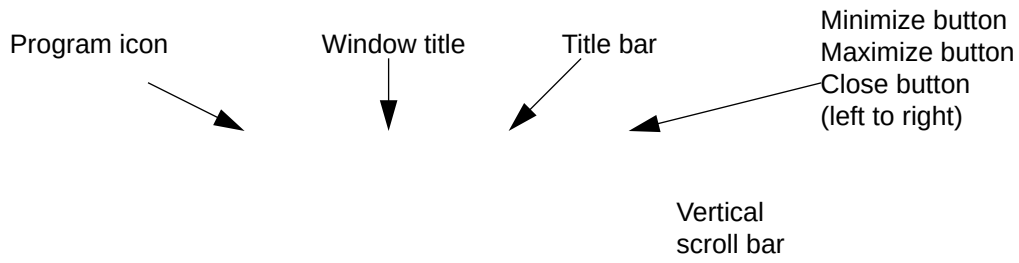
- **The Toploop Window**. Interaction with Lisp can go on through the Toploop window. See chapter 2 in *Programming Tools* for more information.
- **The Status Bar**. Information of interest is printed in the status bar by the system. What is printed often depends on the location of the mouse, or on what you are typing. (Thus, when the mouse is over a button in the toolbar, the status bar displays what clicking on the button will do; and when you are entering a form in a Text Edit window -- like the Toploop Window -- the argument list of the operator in the form is displayed in the status bar.) The status bar can be hidden (and redisplayed) with the F11 key. Note that the status bar is always above any subwindow of `*lisp-main-window*`. Some windows are large enough that the status bar covers part of them. That is when the F11 key is particularly useful. See section 2.7 **The status bar** in *Programming Tools* for more information.

5 Features of Microsoft Windows

You may already be familiar with the information in this section, particularly if you have used MicroSoft Windows for some time. Newcomers to Microsoft Windows may wish to review this section for shortcuts and hints when running Allegro CL for Windows. The illustrations and features are from a Windows 95 machine.

A window

Illustrated below is the Toploop window (which appears when you start Allegro CL for Windows) with some of its features identified:



These are all standard Windows features so we will not discuss them at length. When you press the left mouse button in the Program icon, a menu is displayed allowing you to minimize or close the window (along with other options). Clicking on the Close box or choosing **Close** from the Program icon menu is the same as **Close** in the File menu.

You can expand the window to cover the screen by clicking the Maximize button and make it smaller or into an icon by clicking the Minimize button.

Click above or below the bubble in the scroll bars to scroll the window in the desired direction. Windows can have horizontal as well as vertical scroll bars.

Menus

A good deal of input is done with menus. Whenever a program is running, the menu bar along the top of the screen contains menus appropriate for the program. These menus often have the same names but different items depending on what program is being run. In our documentation, we name menus with a capitalized word in no special font, usually followed by ‘menu’ -- thus the File menu and the Tools menu. Items (also called choices) in a menu are named with their title in **Times Bold**, thus the **Exit** item in the File menu.

Keyboard equivalents to menu items

The following illustration shows the File menu from the Allegro CL menu bar:

Note that each menu and most of the menu items have an underlined letter. This letter indicates how to display and choose from the menu without using the mouse. Hold the Alt key down and press the underlined letter in the menu title (‘F’ in the case of the File menu). Keep the Alt key pressed. Press the letter underlined in the selection you want (‘S’ for **Save** and ‘A’ for **Save As**, for example). The result is the same as selecting the choice from the menu with the mouse.

Note that some choices have a key combination to the right of the choice. Pressing this key combination at any time is equivalent to selecting the menu item with the mouse. For example, the key combination Control-S (Crtl+S in the menu) is equivalent to selecting **Save**.

Note that not all items have keyboard equivalents using the Control key. A few menus and menu items do not have underlined letters indicating an Alt key shortcut (although all the ones illustrated do).

Interrupting

If Lisp seems not to respond, you can interrupt it by pressing the Break key (sometimes labeled Break and Pause). Keep the key pressed for several seconds. It may take Lisp several additional seconds to respond. When it does, it will display a Restarts dialog box similar to the following:

You typically choose **Abort** although you might choose **Enter Debugger** to see if you can determine why Lisp seemed to be hung (e.g. if it was in an infinite loop that does not process Windows events[1]). In any case, you once again have control. (There is also an **Invoke Selected Restart** button, but there are typically only two restarts that correspond to the other two buttons when computation is interrupted.)

What to do if the system hangs

Pressing Break as described just above is the best solution. If that does not work, you can press the combination Ctrl-Alt-Delete which may kill the Lisp process. That is pretty drastic since your work will be lost so only use it as a last resort when pressing Break has clearly failed. (If you are running Windows NT or Windows 95, Ctrl-Alt-Del has a less drastic effect, but you may still not be able to regain control without killing the task.)

1. See the description of **`cg:process-pending-events`** in the Online Help. You should put calls to this function in cpu-intensive loops to ensure you can break out of them if necessary.

The two Enter keys

On most PC keyboards, there are two keys named ‘Enter’. One, next to the alphabetical keys, is typically named ‘Enter ↵’ while the other, usually next to the numeric keypad, is simply named ‘Enter’. These two keys have different codes. In most circumstances, they can be used interchangeably but in some cases they cannot, including the following:

- In a Text Edit window, Enter ↵ is a carriage return while Enter calls for the current Lisp expression to be evaluated. This distinction is particularly important when text is selected (i.e. highlighted). Hitting Enter ↵ replaces the highlighted text with a carriage return while Enter causes the text to be evaluated in the Toploop window.

Dialog boxes

Lisp (like many Windows applications) does some communication via dialog boxes. A Restarts dialog box is displayed when an error occurs, for example -- see the illustration under **Interrupting** above. Note that when a dialog box appears, it often controls the screen and you must choose one of its boxes (**Invoke Selected Restart**, **Abort** and **Enter Debugger** in the illustration above) before continuing with Lisp.

6 Getting information and patches

The World Wide Web page

Franz Inc. maintains a World Wide Web page. Its URL is

```
http://www.franz.com
```

The page contains information about Allegro CL for Windows and other Franz Inc. products. Of particular interest to users of Allegro CL for Windows is the ability to access the Allegro CL for Windows FAQ and patches.

The FAQ (Frequently Asked Questions) is a document written in question and answer format. It is updated regularly, often with answers to questions that the Franz Inc. support staff notices are common. Hints and tips (about optimizing code, for example) are also provided. We recommend that users visit the WWW page from time to time and examine or download the current version of the FAQ.

What is a patch

A patch is a file (typically a *.fsl* file) that, when loaded into Allegro CL, fixes some problem or bug, or (more rarely) adds some feature. Patches are typically produced in response to a report of a bug (that is not trivial and does not have a simple workaround). Note that some features cannot be patched.

Getting patches

Patches are available via the Franz Inc. World Wide Web page. Access the page (whose URL address is given above) and follow directions to Allegro CL 3.0 for Windows patches. You can download patches from the WWW page.

Where to put patch files

Patch files should be in the *UPDATE* subdirectory of the Allegro CL distribution directory (the directory you specified when you installed Allegro CL for Windows). When you install Allegro CL for Windows, an empty *UPDATE* subdirectory is created. Thus, if you installed Allegro CL in the default location, which is *C:\ALLEGRO*, the update directory is *C:\ALLEGRO\UPDATE*.

All patches should be placed in this directory. When Allegro CL starts up, the patches in this directory are read into the image.

When patches are loaded

Patches are loaded when Allegro CL starts up. At that time, all patch files found in the *UPDATE* subdirectory are read into the Lisp image. At the same time a record of what patches are loaded is printed in the banner in the toploop window. Note that saved images do not typically load patches (see the Online Manual entry on **`save-image`** for more information).

Creating an image with patches loaded

You may find it desirable to create an image (with **`save-image`**) that has all the patches loaded. When saving images, though, keep in mind that the whole environment is saved, including any changes introduced by the automatic loading of the startup files, *prefs.lsp* and *startup.lsp*. This means the saved image might have more changes than those introduced by the patches, and it is possible that the interaction between the startup files and the patches could cause Lisp to be in an inconsistent state, leading to mysterious failures. These failures would be hard to debug because suppressing the loading of startup files (a typical thing to do in the face of unexplained behavior) would not be possible because the files are already in the image.

Still, it is desirable to create such images. However, you must do it by hand. Here are the steps. We assume the Allegro CL distribution is in *C:\ALLEGRO*.

1. You should not read any startup files when creating an image with the patches. There are four possible startup files, all located in the *C:\ALLEGRO* directory: *startup.lsp*. *startup.fsl*, *prefs.lsp*, and *prefs.fsl*. Move all such files that exist to a different directory or change their names temporarily. Lisp will now start up without reading any startup file.

2. Start Allegro CL in the usual way. Do not type anything to the Toploop window or choose anything from any menu.
3. Choose **Save Image...** from the submenu displayed by choosing **Images** from the File menu.
4. Supply an image name when asked for one. The name should have the extension *.img*. Do not overwrite *allegro.img* or (if you have the Professional version) *runtime.img*, the image files created when you installed Allegro CL for Windows.
5. The Lisp will appear to restart. At this point, the image file has been created and you can exit from Lisp.
6. Restore any files moved in step 1 to their original location or name.

You may wish to create an icon in the Allegro CL for Windows program group for the saved image. See the Online Manual entry on **`save-image`** for information on creating such an icon, and other ways to initiate a saved image. (The quick and dirty way is to start Allegro CL, choose **Load Image...** from the submenu displayed by choosing **Images** from the File menu, and specify your *.img* file to the dialog that appears.)

> **Note**. An image including patches created as described above will *not* load further patches. If you receive more patches, put them in the *UPDATE*\ subdirectory and create a new image with patches loaded as described above, starting with the original Allegro CL image.

How do you know what patches are loaded in an image

Each patch file from Franz Inc. pushes its description onto `acl:*patches*` when it is loaded. To see what patches are loaded, pretty-print the value of `acl:*patches*`:

```
(pprint acl:*patches*)
```

Brief information on what patches are being loaded is printed in the banner when Allegro CL starts up.

What if a patch needs to be updated?

Sometimes a patch file introduces new problems as well as fixing old ones. In that case, a new patch file (with a later number) is created and the original patch file is replaced with a stub file. For this reason, please observe the following rules:

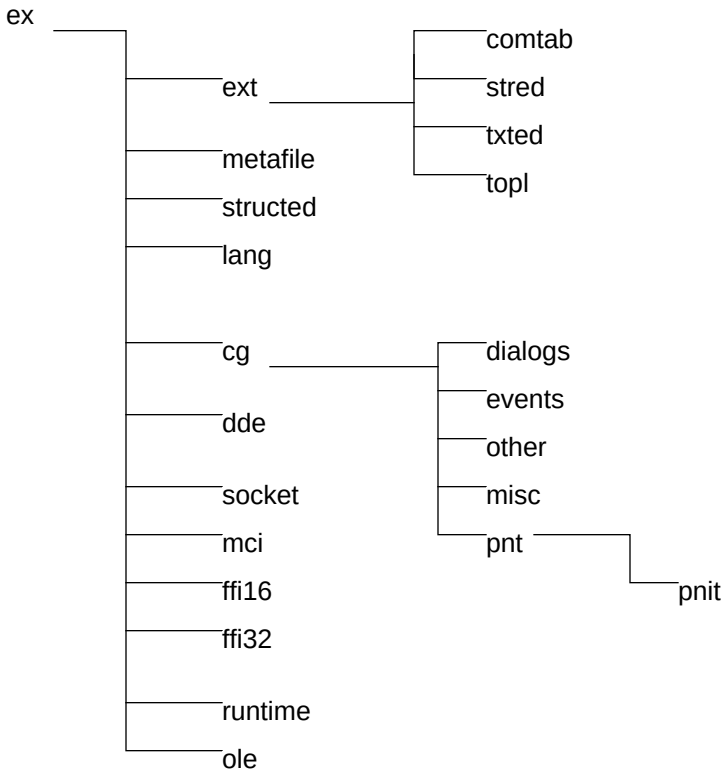
- **Always grab all patches that are available** (including ones you have grabbed before). This will overwrite existing patch files, but that is what is wanted.
- **Always start with the original image** when creating a saved image with all patches loaded. The original image is the image created when Allegro CL is installed. Load all available patches and create a new saved image containing all patches.

7 Example files

Allegro CL for Windows is distributed with numerous examples of Lisp code, illustrating the Common Lisp language and the extensions supplied with Allegro CL for Windows. In this section, we provide a brief guide to these examples.

Where examples are located

All examples are in the *ex* directory included with the distribution. The following diagram shows the various subdirectories of *ex*:



The examples are grouped according to category. The names are fairly descriptive. Here is some more information on the directories.

The *ext* directory

> This directory contains four subdirectories, with examples of comtabs, the text and structure editors, and the Toploop, in the obvious directories.

The *lang* directory

> This directory contains examples from the manual *Lisp Introductory Guide*.

The *cg* directory

> This directory contains subdirectories containing programs illustrating particular features of Common Graphics. The subdirectories have names that describe their contents. *pnt* is an abbreviation for *paint*.

The *ffi16* and *ffi32* directories

> These directories contain examples illustrating the foreign function interface. One is for 16-bit DLL's, which only work in Win32s, and the other is for 32-bit DLL's, which run under both Win32s and Windows NT. See the *Foreign Function Interface* manual.

The *runtime* directory

> This directory contains examples used to build runtime images. The runtime generator is only distributed with the Professional version of Allegro CL for Windows, but the examples are in every distribution. See the *Runtime Generator* manual in the *Professional supplement* if you purchased the Professional version.

The *structed* directory

> The structure editor is no longer part of Allegro CL for Windows. The source files for the structure editor are in this directory and can be loaded into Lisp to provide the functionality available in earlier releases. This is unsupported.

The *dde* directory

> Examples and information on DDE can be found in this directory.

The *socket* directory

> Code and instructions to establish interprocess communication (TCP) using winsock.dll can be found in this directory. These examples will only work if you have

the Professional version of Allegro CL 3.0 for Windows or have purchased the Socket module.

The *metafile* directory

This directory contains a source file (with comments) for Windows Metafile support. This code provided by a Allegro CL 3.0 for Windows customer is included as a service and is not supported.

The *mci* directory

This directory contains unsupported source code for a multimedia interface to Allegro CL 3.0 for Windows.

The *ole* directory

This directory contains an unsupported source code for an OLE interface to Allegro CL 3.0 for Windows.

Note that example files within *lang* are cumulative: that is, later (by number) files depend on code in earlier files. You should compile and load earlier files to be sure the later files will work.

Two text files appear in the *ex* directory: *compiler.txt* and *hints.txt*. They contain useful information about compiler optimization and using Allegro CL for Windows.

8 Anomalies and things to note

In this chapter, we describe features of Allegro CL for Windows which may be different than you might expect. In each case, the behavior is what is intended (see also the *Read This First* document).

- There is a 32K limit on the size of files that can be edited by the text editor (and thus on the size of Lisp source files that can be opened). If you try to open a larger file, the following error will be signaled:

  If you have a file open and add text so the 32K limit is reached, the system will not accept additional text (but will not signal an error). (32K is typically between 500 and 1000 lines of text, depending on the average number of characters per line.) Large files should be broken up into smaller pieces if you wish to use the text editor on them. Note, however, that **`load`** and **`compile-file`** are not affected by the 32K limit.

- The Lisp image will grow as necessary, but only to the limit specified in the *allegro.ini* file included in the distribution (it is located in the directory where the distribution disks were installed -- typically *c:\allegro*). Here are the default contents of the file:

  The maximum heap size is specified by the HeapSize line. The initial value is 16.00 (indicating 16 Megabytes). You may increase this value to allow for larger images, if desired. The other parameters control the behavior of the garbage collector and typically do not need to be changed by users. See the entry on **Storage Use** under the topic **Programming environment** the Online Manual.

- If you initiate stepping with the macro **`step`** in a Toploop window, a Stepper window appears (as you would expect). You cannot type to the Toploop window until stepping has completed, however. This is because the Toploop window is evaluating the form with **`step`** and you must wait until that form completes before new input is accepted. See chapter 8 of the *Programming Tools Introduction* for more information on Stepper windows.

- Common Lisp is for the most part case-insensitive. Except within strings (i.e. characters surrounded by " marks), uppercase and lowercase letters will read the same. Thus, entering `car`, `CAR`, `Car`, or `caR` all denote the symbol `CAR`. The system typically prints in the case specified by the variable `*print-case*`. The initial value of that variable is `:upcase`, so initially the system prints in uppercase. In the documentation, we typically use lowercase, since we believe it is more readable. If you keep in mind that, outside of strings, for most purposes, there is no difference between upper and lowercase, the mixture of upper and lower should not be confusing.

Professional and standard versions are incompatible!

Compiled Lisp (*fsl*) files generated by the Professional version cannot be read into the standard version and *fsl* files generated by the standard version cannot be read into the Professional Version.

Image (*img*) files generated by the Professional version cannot loaded into the standard version (with **load-image**) and cannot be invoked with the standard version *lisp.exe*. Image files generated by the standard version cannot be loaded into the Professional version (with **load-image**) and cannot be invoked by the Professional version *lisp.exe*.

Version 2.0 fsl and img files cannot be used by version 3.0

Compiled Lisp (*fsl*) files generated by the Allegro CL version 2.0 (or 1.0) cannot be read into Allegro CL 3.0 nor can 2.0 or 1.0 image files (*img*) be loaded into Allegro CL version 3.0.

Do not use 2.0 patches including FAQ patches

Some patches (in some cases enhancements) for Allegro CL version 2.0 were distributed in source form, either in the FAQ document or obtained directly from Franz Inc. Do not use these source patches in version 3.0. In most cases the functionality is already present. If it is not, please contact Franz Inc. for information on whether the source patch can be used.

9 Support

Support is available for Allegro CL for Windows. In order to get support, you must first register your purchase with Franz Inc. Instructions for registering your purchase and information on support are included in the Allegro CL for Windows distribution. See the section **Support** in the *Read This First* document for more information. If you have misplaced the *Read This First* document, please write to us at the address below for information on registration and support. Support for Professional version customers is also described in the *Professional* supplement.

Franz Inc.
Allegro CL for Windows Support
1995 University Ave.
Berkeley, CA 94704
USA

510-548-3600
info@franz.com

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

Allegro CL for Windows

Lisp Introductory Guide

version 3.0

October, 1995

Copyright and other notices:

This is revision 1 of this manual. This manual has Franz Inc. document number D-U-00-PC0-03-51017-3-1.

Copyright © 1992-1995 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademark of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

Contents

[This page intentionally left blank.]

PREFACE

This manual *Common Lisp Introduction*, provides an introduction to Common Lisp as a programming language. Users who are unfamiliar with Common Lisp can learn about programming in Common Lisp from this manual. Note, however, that this manual does not describe how to use Allegro CL for Windows, in the sense of describing the user interface. Users familiar with Common Lisp but new to Allegro CL for Windows would do better to start with the *Programming Tools*, which is in volume 2 of the printed documentation.

We are less formal in this manual than in other manuals. We do not concern ourselves particularly with packages and we often provide abbreviate definitions of functions, leaving out, for example, keyword and/or optional arguments not needed to explain the point we are making. Users interested in formal definitions of Lisp functions, variables, macros, etc. should refer to the Online Manual, where all Common Lisp functions, Allegro CL extensions of standard Common lisp functions, CLOS and MOP, and Common Graphics are described. The Online Manual is further described at the end of this preface.

The format for defining functions and other objects is also different in this manual than in the Reference Guides. Here is a typical function definition:

`zerop` tests whether a number is zero. **`zerop`** is clearer and more efficient than the equivalent `(= number 0)`.

The object being described is indented but on the left. The description is in Helvetica type font on the right. Most of the rest of the text is in Times Roman (the same as this paragraph). Examples and code samples are in `Courier`.

There are online code examples supplied with the Allegro CL for Windows distribution. They are in the *ex\lang* directory read off the distribution disks (typically in the *allegro* directory if you followed the default installation instructions). You can try out those examples as you read the text. In general, the example filename reflects the chapter from which the examples are drawn. Note that the examples are cumulative -- that is, examples from later (by number) files may depend on definitions in earlier files.

We have tried to provide in this manual enough information for a user new to Lisp to start programming in Lisp. You may, however, want to supplement this manual with other books about programming in Lisp. There are many on the market. A good general introduction is *Lisp* by Patrick Winston and Berthold Horn (make sure you use the 3rd or later edi-

tion). A widely admired book introducing CLOS is *Object-Oriented Programming in Common Lisp* by Sonya Keene. Both of these books are published by Addison-Wesley of Reading Massachusetts and should be available from any good bookstore selling computer-related books.

The Online Manual

Most of the Common Lisp language, including CLOS, is described in the Online Manual. There is no printed documentation for general Lisp functionality other than the manual you are now reading. The Online Manual is brought up by choosing either **Manual Entry** or **Manual Contents** from the Help menu:

The Online Manual is organized into chapters, and in each chapter, the relevant functionality is defined. If you choose **Manual Entry**, it will open to the definition of the symbol nearest the text cursor if there is one. If there is no such symbol, that fact will be reported in the status bar and the Online Manual will not be displayed. If the nearest symbol does not have a definition in the Online Manual, again that fact will be reported in the Status Bar and the Online Manual will not be displayed. To see the contents of the Online Manual, choose **Manual Contents**.

Chapter 1 Introduction

This guide is designed to introduce Allegro CL for Windows to those who are familiar with conventional programming languages and techniques but have no experience with Lisp. It is not intended to provide an in-depth understanding of Allegro CL for Windows but should give a sound introduction to the language and concepts of symbolic programming.

Lisp (the name is derived from LISt Processing) provides all the features of a general-purpose language but the original objective was to create a language which uses human expressions and which closely resembles human thinking and reasoning. In time, many 'dialects' have evolved, but efforts have been made to combine the best features from these, resulting in the definition of an industry standard known as Common Lisp. Allegro CL for Windows adheres strictly to this standard while offering many additional enhancements including powerful graphics capabilities and an integrated programming environment.

Many of the early criticisms of Lisp concerning speed, size, and complexity have been successfully addressed as the language has developed and it is hoped that you will find Allegro CL for Windows a powerful, efficient and flexible development tool for all applications.

Please note that unless otherwise stated, all comments made about Lisp will refer to both Common Lisp and Allegro CL for Windows.

1.1 Getting started

Lisp is a language which demands precise syntax, since it makes no distinction when storing between data and program. It uses terminology which may be familiar, totally new, or have different meanings in conventional languages. Other publications about the language introduce words or phrases which serve only to add to the confusion. We have therefore striven to keep these to a minimum throughout the text, but we have included the more common ones in a glossary in the appendix.

Most programs are called *functions* and there are many useful functions provided by Lisp. To *call* a function, a typical format would be:

```
(fn a1 a2 ... an)
```

where **fn** is the function name and a1 ... an are the arguments.

Consider this simple example:

```
(+ 3 2 5)
```

where **+** is the function of addition and 3, 2, and 5 are the arguments. 10 is the value returned.

Unlike conventional mathematical notation, the function (in this case, addition) appears first. It is also possible to combine functions but it is important to remember that **Lisp always evaluates the arguments before applying the function**. Using another simple mathematical example, we can subtract two numbers and divide the result as a single operation:

```
(/ (- 28 4) 2)
```

In this example, the argument `(- 28 4)` is *evaluated* first and the returned value is then divided by 2. Note that the argument `(- 28 4)` is in itself a function call and therefore follows the function syntax format.

This is how the Top Loop window should appear on the screen. (You will see an actual date and time, of course, but they were determined after this manual went to the printers.)

So far we have used mathematical examples, but Lisp's greatest strength lies in its ability to manipulate symbolic data which has no mathematical context. This will become clear in the following chapter. It will be helpful if you try out the examples as we go along. You can read the example files into a text edit window. You can evaluate the forms in the examples by placing the cursor at the beginning of the form and pressing Alt-Enter (equivalently, pressing the Enter key next the numeric keypad). You can also type the examples in directly. In that case, input may be followed by a carriage return, that is, the Enter key next the alphabetic keyboard, thus allowing you to lay out your programs to make them easier to read. Lisp will not try to execute until it sees a complete form and then the form will be evaluated.

[This page intentionally left blank.]

Chapter 2 Basic data manipulation

In Lisp there are three basic types of data which are known as *symbols*, *numbers*, and *lists*.

- Symbols are named data objects. For example, `dog`, `cat` and `man` are symbols. They may also be functions and variables used within programs and have some other features which are explained later.
- Numbers fall into four categories: Integers, Floating-point, Ratios and Complex. 13, -56.2, 22/7 and #C(0 2) are examples of each of these categories of numbers: #C(0 2) is the square root of -4, namely 0 + 2*i*.
- A list is a series of elements enclosed in parentheses. Each element may be a symbol, number or another list, thus nested structures can be constructed to represent complex data. `(dog bites man)` is a list of three elements.

In Lisp, an item of data can be evaluated to yield a value. In this context, such an item is referred to as a *form*. As we shall see later, Lisp programs are simply collections of forms to be evaluated. Some forms in the following examples may be preceded by a quote mark (`'`) which indicates that the form is not to be evaluated; it is a literal expression. The quote mark will be discussed later in more detail, as will **`setf`** which also appears below.

2.1 Defining lists

Let us assume that we wish to create a list of things that we like. Remembering that the elements must be enclosed in parentheses, our list could be:

```
(lasagne claret ferraris mozart summer)
```

In order to refer to this list without typing it out in full each time, a name can be given to it such as `likes` and this assignment would be done as follows:

```
(setf likes
      '(lasagne claret ferraris mozart summer))
```

to which Lisp would respond:

```
(LASAGNE CLARET FERRARIS MOZART SUMMER)
```

`likes` now has the list as its value, and the list is returned as the value of the **`setf`** form. In this example, we are not primarily interested in the returned value, but in the assignment which took place. The assignment is described as a side effect of the evaluation.

> **Note**: Many forms, especially mathematical ones, have no side effects and are evaluated simply to return a value. However, side effects should always be borne in mind, as unexpected ones can cause bugs.

The contents of any expression are displayed by typing its name. For instance,

```
likes
```

would return:

```
(LASAGNE CLARET FERRARIS MOZART SUMMER)
```

In a similar way, we can assign the things we dislike to another list called `dislikes`, and display the contents as follows:

```
(setf dislikes
     '(taxation cocktails skiing rain
       (hairy spiders) work))

→ (TAXATION COCKTAILS SKIING RAIN (HAIRY SPIDERS) WORK)

dislikes
→ (TAXATION COCKTAILS SKIING RAIN (HAIRY SPIDERS) WORK)
```

In this example, we not only have a list of individual elements we dislike, but we also have a list as an element of the list, namely `(hairy spiders)`. All the elements are top-level elements while `hairy` and `spiders` are second level elements.

2.2 Adding to and removing from lists

Having defined our lists, we may wish to add or remove elements. If, for example, `skiing` has been assigned to the wrong list, it must be removed from `dislikes` and added to `likes`. Lisp provides some built-in functions which allow us to do this.

`cons` The **`cons`** function adds an item to the front of a list. It takes two arguments: the second argument must be a list, but the first may be any element.

```
(setf likes (cons 'skiing likes))
→ (SKIING LASAGNE CLARET FERRARIS MOZART SUMMER)
```

`skiing` is added to the elements stored in `likes` and the result is assigned back to `likes`.

`remove` The function **`remove`** deletes an item from a list. Like **`cons`**, it accepts two arguments, the second of which must be a list.

```
(setf dislikes (remove 'skiing dislikes))
→ (TAXATION COCKTAILS RAIN (HAIRY SPIDERS) WORK)
```

`skiing` is deleted from the elements stored in `dislikes` and the result is assigned back to `dislikes`.

> **Note**: In both these examples the functions **`cons`** and **`remove`** do not affect the original lists: they create temporary lists. In order to make these changes permanent, **`setf`** is used to assign the temporary lists to the original ones.

2.3 Introduction to defining functions

In order to define a function, Lisp needs to know three things:

1. The name of the function.
2. The number of arguments it needs.
3. The task the function is to perform.

To combine the two steps in our previous example, we can write a simple function to which we will give the name **`newlikes`**. This is the name that will be used each time we wish to call this function.

`defun` This macro lets you define your own functions.

```
(defun newlikes (name)
   (setf likes (cons name likes))
   (setf dislikes (remove name dislikes)))
→ NEWLIKES
```

In the above example, the three requirements are met as follows:

1. The name of the function is **`newlikes`**.
2. There is one argument, the variable *`name`*.
3. The function takes the value of the variable *`name`*, adds it to the list `likes` and removes it from the list `dislikes`.

Note: Even though the **`defun`** form is defining a function, it is still evaluated and, in this case, returns the name of the function, **`newlikes`**.

If we now wish to call our function to transfer cocktails from `dislikes` to `likes`, we do so in this way:

```
(newlikes 'cocktails)
→ (TAXATION RAIN (HAIRY SPIDERS) WORK)
```

`newlikes` returns the modified list `dislikes` since this is the value returned by the last form within the function.

`newlikes` now joins the built-in functions[1] (**`cons`**, **`remove`**, **`+`**, **`/`**, etc.) and becomes another function we can call at any time. It is in every way the equal of these functions. Unlike some other programming languages, there is nothing special or "magic" about built-in functions.

1. In some contexts, built-in functions are referred to as *primitives*.

2.4 Inserting comments

In any programming environment, meaningful comments are invaluable to the programmer: they make the code easier to understand and therefore easier to modify or debug. The start of a comment in Lisp is indicated by a semicolon, and all text following it to the end of the line is ignored.

```
(defun newlikes (name)
   ;add name to start of likes
   (setf likes (cons name likes))
   ;delete name from dislikes
   (setf dislikes (remove name dislikes)))
→ NEWLIKES
```

Throughout this guide we have tried to make the examples self-explanatory and comments are therefore only given to clarify particular points. However, we recommend that you include as many comments as necessary to make your own code easier to understand. Remember that although you know exactly what a program does when you write it, you will probably have forgotten in six months' time when you look at it again.

2.5 The super parenthesis

The above function **`newlikes`** requires three parentheses at the end to terminate the function correctly. This is made easy for you with the automatic flashing of parentheses which match. However, as you begin to write more complex functions, you could require five or more concluding parentheses. Rather than relying on the flashing of matching parentheses, you can finish with a single 'super-bracket': a right-hand square bracket (]). This instructs Lisp to terminate all matching parentheses. You may wish to try this on some of the examples.

[This page intentionally left blank.]

Chapter 3 Arithmetic

Before looking in more detail at the other functions which are available to manipulate lists, let us examine the mathematical aspects of Lisp. As stated earlier, Lisp does not follow conventional mathematical notation but maintains uniformity in the syntax of all function calls: the required function is always the first element of the call. In all the examples below, the value returned by Lisp follows each function call and explanations are given only where relevant.

3.1 Performing calculations

+ The addition function.

```
(+ 3.1 2.7) → 5.8
(+ 3.1 2.7 100.1) → 105.9
(+ 4 7) → 11
(+ 4.7 1.1) → 5.8
```

Note that 9/2 and 17/2 are ratios. Unlike most other programming languages, Lisp allows you to use fractions as well as integers and floating point numbers.

```
(+ 9/2 4) → 17/2
```

- The subtraction function.

```
(- 3.1 2.7) → 0.4
(- 3.1 2.7 100.1) → -99.7
(- 1 3 5 7) → -14
(- 4) → -4
(- -4) → 4
```

***** The multiplication function.

```
(* 3.1 7.0) → 21.7
(* 3.1 7.1 10.0) → 220.1
(* 4 3) → 12
(* 4.0 3) → 12.0
```

/ The division function.

```
(/ 21.7 7.0) → 3.1
(/ 21.0 3.0) → 7.0
(/ 21 3) → 7
(/ 21 3.0) → 7.0
(/ 9 2) → 9/2
(/ 9.0 2) → 4.5
(/ 9.0 2.0) → 4.5
```

If more than two arguments are supplied, the first argument is divided by each of the other arguments in turn.

```
(/ 30 2 3) → 5
(/ 139 7 2) → 139/14
```

If only one argument is supplied, the result is the reciprocal of that argument.

```
(/ 2.0) → 0.5
(/ 2) → 1/2
```

sqrt The square root function.

```
(sqrt 4) → 2.0
(sqrt 4.0) → 2.0
```

If the argument is negative the result is not an error, but a complex number. In Lisp, complex numbers are displayed in the form #C(3 2) which is equivalent to 3 + 2i.

```
(sqrt -4) → #C(0.0 2.0)
(sqrt (/ -1 49)) → #C(0.0 0.142857)
(+ 2 #C(0 1/7)) → #C(2 1/7)
```

expt The exponential function.

```
(expt 2 10) → 1024
(expt 10 2) → 100
(expt 2/3 3) → 8/27
(expt -4 1/2) → #C(1.22514845490862E-16 2.0)
```

If either argument is a floating point number, the result is a floating point number and so an approximation may occur. In this case 0.0 is correctly approximated by 1.22514845490862E-16.

log The **log** function. The base defaults to *e*, unless another base is specified by giving it as a second argument.

```
(log 1)→ 0.0
(log 10) → 2.3025
(log 10 2) → 3.3219
(log 238) → 5.4723
```

abs The absolute value function.

```
(abs 8) → 8
(abs -8) → 8
(abs 8.9) → 8.9
(abs -8.9) → 8.9
```

truncate The truncate function, which rounds towards zero. Note that **truncate** returns multiple values. The first is the argument truncated: the second is the argument supplied minus the first value returned.

```
(truncate 13.6) → 13 0.6
(truncate 0.7) → 0 0.7
(truncate -0.7) → 0 -0.7
(truncate -1.7) → -1 -0.7
```

round The round function, which rounds to the nearest integer. Note that **round** returns multiple values. The first is the argument rounded to the nearest integer: the second is the argument supplied minus the first value returned.

```
(round 3.1) → 3 0.1
(round 3.7) → 4 -0.3
```

If the argument is exactly midway between two integers, the first value returned is the nearest even integer.

```
(round 3.5) → 4 -0.5
(round 4.5) → 4 0.5
(round 5.5) → 6 -0.5
```

rem The remainder function, which returns the remainder when the first argument is divided by the second.

```
(rem 17 4) → 1
(rem 4 17) → 4
(rem -17 4) → -1
```

float The **float** function, which converts integers into floating point numbers.

```
(float 4) → 4.0
(/ (float 7) (float 4)) → 1.75
```

3.2 Testing numbers

Lisp provides a number of functions to perform tests which are called predicates. Technically, a predicate is a function that returns one of two values: True or False. These are normally indicated by the constants `nil` (False) and `t` (True). However, any non-`nil` value also indicates that a test is true and is often of more use if the result of the test is to be used in further computations. The predicates listed below can be used on numbers and symbols which have numeric values. This is demonstrated by defining the symbols zero, one, two, three and four to represent their numeric equivalents and using them throughout the examples.

```
(setf zero 0 one 1 two 2 three 3 four 4) → 4
```

Note: As we have used a single form to assign a number of values, the form returns only the last value assigned.

> The descending order function, as its name suggests, tests that all the arguments supplied are in descending order.

```
(> 100 10 one 0.1) → T
(> 10 100 1 0.1) → NIL
(> 100 10 four zero) → T
```

< The ascending order function tests that all the arguments supplied are in ascending order.

```
(< 1 3 5 7 11 9) → NIL
(< 1 three 5 7 9 11) → T
```

max returns the largest (most positive) argument.

```
(max 1 2 3 4 5 6 7) → 7
(max (* 1 7) (* 2 6) (* 3 5)) → 15
(max (expt 2 10) (expt 10 2)) → 1024
```

min returns the smallest (most negative) argument.

```
(min 1 2 3 4 5 6 7) → 1
(min (expt 2 10) (expt 10 2)) → 100
```

= tests whether numbers have the same numerical value, regardless of their type.

```
(= (/ 2 3) (/ 4.0 6.0)) → T
(= (/ two three) (/ 4.0 6.0)) → T
(= 3.141592653 (/ 22 7)) → NIL
```

evenp tests whether an integer is even.

```
(evenp 17) → NIL
(evenp (expt 3 4)) → NIL
(evenp two) → T
(evenp (* three 17)) → NIL
```

oddp tests whether an integer is odd.

```
(oddp 17) → T
(oddp (expt 3 4)) → T
(oddp two) → NIL
(oddp (* three 17)) → T
```

minusp tests whether a number is negative.

```
(minusp 22) → NIL
(minusp -13) → T
(minusp (- three)) → T
```

zerop tests whether a number is zero. **zerop** is clearer and more efficient than the equivalent `(= number 0)`.

```
(zerop (* 2 0)) → T
(zerop (+ 2 0)) → NIL
(zerop zero) → T
(zerop (- four three one)) → T
```

Chapter 4 List manipulation

4.1 Extracting elements

The following functions allow us to extract elements from lists, either as individual elements or as parts of the list. It should be noted that none of these functions affect the contents of the original list in any way.

first returns the first top-level element of the specified list. The list may be supplied as a literal or by its symbolic name.

```
(first '(cocktails skiing lasagne claret ferraris mozart summer))
→ COCKTAILS
;Recall in chapter 2 that likes was the name given to the list
;'(cocktails skiing lasagne claret ferraris mozart summer).
(first likes)
→ COCKTAILS
```

rest is the complement of **first**. It returns the list without the first element.

```
(rest '(cocktails skiing lasagne claret ferraris mozart summer))
→ (SKIING LASAGNE CLARET FERRARIS MOZART SUMMER)
(rest likes)
→ (SKIING LASAGNE CLARET FERRARIS MOZART SUMMER)
```

car and **cdr** are identical to **first** and **rest** respectively. These functions are remnants from the original implementation of Lisp which have been superseded by **first** and **rest**; they are not mnemonic and as such, more difficult to remember. However, they are still prevalent in many programs and pub-

lications about Lisp and you should therefore be familiar with them and their operation.

Several **car**s and **cdr**s are needed to find an element buried deep in a list. All the combinations of up to four **car** and **cdr**s are defined as separate Lisp functions. The names of all these functions begin with *C* and end with *R* and in between is a sequence of *A*s and *D*s corresponding to the composition performed by the function (**c***xx***r** .... **c***xxxx***r**). For example, these mean the same:

```
(cadr likes) ≡ (car (cdr likes))
(cadddr likes) ≡ (car (cdr (cdr (cdr likes))))
```

(≡ is not a Lisp expression but in this manual means "is equivalent to".).

second to **tenth** are nine other functions provided by Lisp to complement **first** and **rest** and enable each of the first ten elements to be extracted from a list. If we remember that **first** is the same as **car**, it follows logically that **second** is the same as **cadr**, **third** is the same as **caddr**, and so on.

```
(fourth likes)
→ CLARET
(tenth likes)
→ NIL
```

In this example, there is no tenth element in the list `likes`, so the value returned is `nil` which is also the empty list ().

nth returns the specified element of a list where **first** (**car**) is element zero of the list. It takes two arguments: an integer and a list. The integer must have a non-negative value. **nth** overcomes the limitations of **second** to **tenth** as the integer can represent any element, no matter how long the list. Consider the following:

```
(nth 0 likes) ≡ (first likes) ≡ (car likes)
(nth 3 likes) ≡ (fourth likes) ≡ (cadddr likes)
```

If the integer is greater than the length of the list, `nil` (the empty list) is returned.

last takes a single argument (which must be a list) and returns a list with one element which is the last item of the specified list. If the list is empty, `nil` is returned.

```
(last likes)
→ (SUMMER)
```

Before we leave this section, it may be helpful to explain a bit more about the role of the quote mark in Lisp syntax. As said earlier, Lisp adheres strictly to the principle that the function call is always the first element after the opening parenthesis. It also assumes that everything which follows is to be evaluated unless told not to. Let us look at an example:

```
(first (rest likes))
```

tells Lisp to perform **rest** on the value of `likes` to produce:

```
(SKIING LASAGNE CLARET FERRARIS MOZART SUMMER)
```

and then to apply **first** to this result to produce:

```
SKIING
```


If we have not given our list a name, we can write the list out in full but it must be preceded by a quote mark or Lisp will not recognize it as data. If we write:

```
(first (rest (cocktails skiing lasagne claret
             ferraris mozart summer)))
```

Lisp does not know otherwise so it treats `(cocktails....)` as a function call and `skiing`, `lasagne`, etc. as variables. This would result in an unbound variable error. By inserting a quote mark in front of this list, Lisp immediately knows that this is literal data and does not try to evaluate it.

```
(first (rest '(cocktails skiing lasagne claret
               ferraris mozart summer)))
→ SKIING
```

quote is a special function which takes a single argument and returns that argument without evaluating it. The quote mark is a shorthand form for **quote**. Thus:

```
'(a b c) ≡ (quote (a b c))
```

4.2 Substituting elements

So far we have seen how to make lists and how to extract individual list elements. Let us now look at a way to replace elements within lists.

subst The **subst** function substitutes one element in a list for another. Unlike the other functions we have used so far, **subst** works through each level of the list to find every occurrence of the specified element. It takes three arguments, the last of which must be a list. The other arguments may be symbols, numbers or lists. **subst** is called in the following format:

```
(subst n o l)
```

where

n is the new argument.

o is the old argument.

l is the list containing the old argument.

For example, if x has the value `(y z)`:

```
(subst 'y 'z x)
→ (Y Y)
(subst 'z 'a '(a b (a b b a) b a))
→ (Z B (Z B B Z) B Z)
```

4.3 Combining lists

There are three main functions which can be used to combine lists: **cons**, **list** and **append**. Although these may appear to do the same job, they produce very different results and before proceeding it is important to understand the ways in which they differ. Choosing the right one may be very important later.

cons The function **cons** adds an item to the front of a list (see also Section 2.2). It takes two arguments, the second of which

must be a list. The first can be a symbol, number or list. The function returns a new list with the first argument as the first element.

```
(cons 'a '(b c d))
→ (A B C D)
(cons '(a b) '(c d))
→ ((A B) C D)
```

list The function **list** also makes a new list from the arguments but the elements of the list are not merged. It can take any number of arguments which may be symbols, numbers or lists.

```
(list 'a '(b c d))
→ (A (B C D))
(list '(a b) '(c d))
→ ((A B) (C D))
```

append The function **append** creates a new list by merging two or more lists into a single list. It takes two or more arguments, each of which must be a list. The argument lists are not changed.


```
(append '(a b) '(c d))
→ (A B C D)
(append '(a (b c)) '((d e) f))
→ (A (B C) (D E) F)
```

In order to clarify this, let us look at the way the three functions operate on the same data.

```
(setf x '(y z)) → (Y Z)
(cons x x) → ((Y Z) Y Z)
(list x x) → ((Y Z) (Y Z))
(append x x) → (Y Z Y Z)
```

It is well worth taking the time to make sure you are absolutely clear about the differences between these three functions as you could end up with a totally different result from the one intended.

[This page intentionally left blank.]

Chapter 5 Introducing input and output

5.1 print and read

Simply, input and output in Lisp are handled by the functions **read** and **print** respectively. Although you have not been aware of it, you have seen these in action already when you have tried out the examples. User interaction is handled by the Toploop. Its purpose is:

1. To accept and print to the screen the code you type in.
2. To evaluate it.
3. To print the returned value.

It uses **read** and **print** to do this. **read** and **print** are obviously interactive.

print The **print** function accepts one argument. It prints the value of the argument. However, when the expression is evaluated, the Toploop also prints the returned value. This gives the appearance of the same thing being printed twice but these two operations are not the same: a value which is returned can be used by another function; a value which is printed cannot. Look at the following example:

```
;Recall in chapter 2 that dislikes was defined as the list
;'(taxation rain (hairy spiders) work).
(print (first dislikes))
TAXATION
→ TAXATION
```

Notice that TAXATION is printed twice: as the value of `(first dislikes)` and as the value returned after it has been evaluated. TAXATION could, if required, be used as an argument of another function such as **`newlikes`**. We can demonstrate this with another simple example:

```
(cons 'cheese (print (rest dislikes)))
(RAIN (HAIRY SPIDERS) WORK)
→ (CHEESE RAIN (HAIRY SPIDERS) WORK)
```

Here, `(RAIN (HAIRY SPIDERS) WORK)` is the value printed by **`print`** while `(CHEESE RAIN (HAIRY SPIDERS) WORK)` is the value returned by the **`cons`** function. Without **`print`**, the intermediate step would not have been printed. Printing is a side effect of evaluating **`print`** and the output is always followed by a space.

`read` The function **`read`** does not accept any arguments. When called, it causes Lisp to pause while it monitors the keyboard and accepts whatever the user types. The input becomes the value of **`read`** but it is not evaluated. There is no indication that **`read`** has been called and it is therefore usual to precede a **`read`** call by printing a suitable prompt. Let us write a function which takes four input numbers and works out the average. Note in this example that even when a function has no arguments, we must still include the empty list as its argument list.

```
(defun meanfour1 ()
   (/ (+ (read) (read) (read) (read)) 4))
→ MEANFOUR1
```

To call this function, one must type (meanfour1) followed by enter, then each of 2, 4, 6 and 8 followed respectively by enters. The screen will appear as:

```
(meanfour1)2468
→ 5
```

The four input values are added together and then divided by 4 to produce the returned value, 5.

Later on, it may be possible for you to write your own **read** and **print** functions. However, it is worth noting that you must be very careful about naming your functions, as you can overwrite existing functions if the name has been used before. Should you try to write your own system functions, it is safer to precede the name with an identifier. For example, **my-read**, **my-cons**, etc. (Common Lisp also has quite a sophisticated "package" system, which allows the user to put his or her functions in a separate name space, avoiding this kind of conflict.)

5.2 eval

As well as **print** and **read**, the Toploop also uses a function called **eval**. Obviously, the correct sequence is `(print (eval (read)))`.

eval accepts one argument which it evaluates. **eval** is how Lisp runs programs. When you **eval** an object, you are converting data into program. Thus:

```
(eval 23) → 23
(eval nil) → NIL
(setf fred 63) → 63
(setf y 'fred) → FRED
(eval y) → 63
(eval 'y) → FRED
(eval ''y) → Y
(eval '''y) → 'Y
(eval '(first (rest '(x y z)))) → Y
(eval (first (rest '(x y z)))) → FRED
```

eval is not often used in code because all the arguments to function calls are evaluated automatically. When using **eval**, a frequent mistake is to force too many evaluations.

5.3 Printing special characters

Certain characters have special meanings to **`print`** and **`read`**. Spaces, quotes and brackets are taken as list delimiters and lowercase letters in symbols are converted into uppercase letters by **`read`**. Sometimes it is desirable to suppress this special treatment. You would have to do this if you wanted to create a symbol containing spaces or lowercase letters, or one which would otherwise be taken as a number.

To introduce a single special character into a symbol, precede it with a backslash character (\), known as the single escape character. If, for example, we wish to use the phrase "`annual (net) income`" as a single symbol, we could do this as follows:

```
(setf asymbol 'annual\ \(net\)\ income)
→ ANNUAL\ \(NET\)\ INCOME
```

Notice that the lowercase letters we typed in the symbol have been converted to uppercase.

The backslash indicates that the following single character is to lose any special meaning it may have. This is clumsy when there are many characters to suppress, as in the above example. An easier way to do this is to surround the whole symbol with vertical bars, known as multiple escape characters.

```
(setf another-symbol '|ANNUAL (NET) INCOME|)
→ ANNUAL\ \(NET\)\ INCOME
```

Notice that we had to type the whole symbol in uppercase this time and also that it was still printed out in the single escape format. This ilustrates an important point - once a symbol has been read in, it forgets all about exactly how it was typed in. So, as far as Lisp is concerned,

```
ANNUAL\ \(NET\)\ INCOME ≡ |ANNUAL (NET) INCOME|
```

However, note that (because of case differences):

```
ANNUAL\ \(NET\)\ INCOME
```

is not the same as:

```
|annual (net) income|
```

5.4 Derivatives of print

There are derivatives of **print** available to assist in the presentation of output. Below are listed three of these which you may find useful.

terpri The function **terpri** forces a carriage return which has the effect of making the output begin on the next line. It does not take any arguments.

prin1 The function **prin1** is like **print** but it does not start with a new line nor does it print a space after the output.

princ The function **princ** is like **prin1** except that **princ** does not include any quotation marks or vertical bars which are included in the arguments. **princ** prints only the argument without any spaces or carriage returns.

The following example demonstrates the use of **princ** and **terpri** within a function to format the output:

```
(defun meanfour2 (first second third fourth)
   (terpri)
   (princ "The average of")
   (terpri)
   (princ first)
   (princ " ")
   (princ second)
   (princ " ")
   (princ third)
   (princ " ")
   (princ fourth)
   (terpri)
   (princ "is")
   (/ (+ first second third fourth) 4))
```

```
→ MEANFOUR2
```

If you now type:

```
(meanfour2 2 4 6 8)
```

you should see:

```
The average of
2 4 6 8
is
5
```

Chapter 6 Binding and scope

6.1 Free and bound variables

So far, whenever we have wished to assign a value to a variable, we have used **setf**. This "fixes" the value which then remains until a new assignment is encountered. We may wish to use a variable within a section of code but not know if it has been preset to a different value elsewhere. Fortunately, we can give a variable a temporary value within a part of a program using **let**. As soon as we leave the section of code governed by the **let**, the temporary value is discarded and the original value is restored. The process of assigning a specific value within a particular section of code is known as binding. If we use a variable within a function without establishing a new binding, that variable is described as being free (or unbound) in relation to the function. Obviously, if we need a temporary variable, it is wise to use **let** to create a new binding to guarantee that we do not trample on our own, or anyone else's, variables.

let creates a new binding for a variable or variables within a function or section of code and assigns a value to it. On leaving the **let** form, the binding is discarded and the original binding and value are restored.

```
(setf day 'monday)
→ MONDAY
(let((day 'tuesday))
  (print day))
TUESDAY
→ TUESDAY
(print day)
MONDAY
→ MONDAY
```

```
(setf x 5)
→ 5
(setf y 7)
→ 7
(let ((x 10))
  (* x y))
→ 70
x
→ 5
```

Remember that values are printed once by **`print`** and again as the value of the expression.

In the second example using **`let`** above, *x* is *bound* by the **`let`**, whereas *y* is *free*.

In Chapter 5, we wrote a function (**`meanfour2`**) which calculated the average of four numbers. The numbers were included as arguments to the function but we will now see how we can use **`let`** to allow us to accept the numbers from the keyboard using **`read`**.

```
(defun meanfour3 ( )
   (let ((total 0))
      (terpri)
      (princ "Enter the first value: ")
      (setf total (+ total (read)))
      (terpri)
      (princ "Enter the second value: ")
      (setf total (+ total (read)))
      (terpri)
      (princ "Enter the third value: ")
      (setf total (+ total (read)))
      (terpri)
      (princ "Enter the fourth value: ")
      (setf total (+ total (read)))
      (terpri)
      (princ "The average is ")
      (princ (/ total 4))))
→ MEANFOUR3
```

If you use **`meanfour3`** to calculate the average of 2, 4, 6 and 8 you will see the following:

```
(meanfour3)
Enter the first value: 2
Enter the second value: 4
Enter the third value: 6
Enter the fourth value: 8
The average is 5
→ 5
```

Be careful if any of your variable bindings refer to other bindings in the **let**: all the values are calculated before making the bindings (i.e. “in parallel”).

```
(setf a 1)
→ 1
(let ((a 2) (b (+ a 1)))
   b)
→ 2
```

Note that 2 (rather than 3) is the value returned. The function **let*** permits binding in series. See the entry on **let*** in Allegro CL Online Help for more information.

Before leaving this section, note that **defun** also creates a new binding for any variables within the function. You can bind several variables at the same time with **let**: simply include more variable-value pairs as shown below:

```
(let ((a 0) (b 1) (c 2) (d 3))
   (+ a b c d))
→ 6
```

6.2 Scope

This brings us to another consideration in Lisp known as scope. The *scope* of a variable binding is the range of the program in which the variable can be used to access a particular value. Lexical scoping (sometimes called textual) is the normal sort of scoping in Common Lisp: the value bound to a variable is determined by the textual context in which the variable is used.

```
(setf x 5)
→ 5
```

```
(defun print-x ()
   (print x))
→ PRINT-X
(let ((x 10)) (print x) (print-x))
10
5
→ 5
```

The free variable *x* in the function **print-x** is bound to the value 5, even when the function is called after *x* has been lexically bound to 10 in the **let**. *x* has the value 10 when used within the **let**: it is the textual context that determined the binding. Lexical scoping in Lisp is exactly the same as in most other block-structured languages, such as Pascal or Modula-2. The alternative to lexical scoping is dynamic scoping. Here the value bound to a variable takes account of any bindings which may have been made dynamically, i.e. as the program is executing. We will repeat the same example using dynamic scoping, but to do this, we need to use a special variable.

Lisp variables which use dynamic scoping are known as *special variables*. Since their use can lead to obscure and difficult bugs, most Lisp programmers use them only sparingly, if at all. We mention them here only to show the difference between the two types of scoping. You should consult a more advanced text for details of how to use them.

You can make a special variable with **defvar**: this declares the variable to be special wherever it is used. It is a good idea to give special variables distinctive names so that you do not re-use them accidentally. A common practice is to surround the name with *'s.

```
(defvar *y*)
→ *Y*
(setf *y* 5)
→ 5
(defun print-y ()
   (print *y*))
→ PRINT-Y
(let
    ((*y* 10))
   (print *y*)
   (print-y))
10
10
```

```
→ 10
*y*
→ 5
```

Notice that the **print-y** function now picks up the value of *y* bound by the **let**, rather than the global value. This binding was the one most recently executed.

[This page intentionally left blank.]

Chapter 7 Conditionals

So far, we have seen how to define functions and write small programs. But, as in other languages, without an ability to choose between alternatives, our programs will be limited in their operation. This chapter covers different aspects of decision making in Lisp.

7.1 Testing symbols

In Section 3.2 we defined a predicate as a function which returns a value of True or False and looked at some which operated on numbers or symbols with numeric values. We will now look at predicates which perform tests on symbolic rather than arithmetic data.

symbolp tests whether its argument is a symbol.

```
(symbolp 'lasagne) → T
;Recall in chapter 2 that likes was defined as the list
;'(cocktails skiing lasagne claret ferraris mozart summer).
(symbolp likes) → NIL
(symbolp '(taxation rain work)) → NIL
(symbolp nil) → T
```

listp tests whether its argument is a list.

```
(listp likes) → T
(listp 'lasagne) → NIL
(listp '(taxation rain work)) → T
(listp nil) → T
```

numberp tests whether its argument is a number.

```
(numberp 6) → T
(numberp 'lasagne) → NIL
(setf five 5) → 5
(numberp five) → T
(numberp 'five) → NIL
```

equal tests whether its two arguments are the same. There are other ways of doing this, but these will be discussed later. Notice that two numbers are considered to be **equal** if they have the same value and are of the same type.

```
(equal five 5) → T
(equal 5 5.0) → NIL
;Recall in chapter 2 that dislikes was defined as the list
;'(taxation rain (hairy spiders) work).
(equal dislikes '(taxation rain (hairy spiders) work)) → T
```

null tests whether its argument is an empty (null) list and returns `t` if it is.

```
(null '()) → T
(null dislikes) → NIL
```

member tests whether an object is an element of a list. It accepts two arguments, the second of which must be a list; the first may be a symbol or another list. If the first argument is not present, **member** returns `nil`, otherwise it returns the fragment of the list that begins with the first argument. This is useful if the result is to be used in further computations.

```
(member 'rain likes) → NIL
(member 'rain dislikes) → (RAIN (HAIRY SPIDERS) WORK)
(member 'spiders dislikes) → NIL
```

In the last example, `spiders` is not found because it is not an individual element. **member** does not identify objects which are buried within lists.

7.2 Logical operators

As well as the predicates discussed, Lisp provides the logical functions **and**, **or** and **not**. When successful, **and** and **or** return the last value evaluated which is also useful for further computations.

and takes one or more arguments which it evaluates from left to right. If any argument returns `nil`, the **and** returns `nil` and any remaining arguments are not evaluated. If no argument evaluates to `nil`, the value of the last argument is returned.

```
(and (member 'claret likes) (member 'rain dislikes))
→ (RAIN (HAIRY SPIDERS) WORK)
(and (member 'work likes) (member 'rain dislikes))
→ NIL
(and (evenp 8) (numberp 6) (listp likes))
→ T
```

or takes one or more arguments which it evaluates from left to right. If any argument returns a non-`nil` value, the **or** returns the value and any remaining arguments are not evaluated.

```
(or (evenp 9) (member 'rain dislikes))
→ (RAIN (HAIRY SPIDERS) WORK)
```

not returns `t` if its argument returns `nil` when evaluated, otherwise it returns `nil`.

```
(not (member 'rain likes)) → T
(not (member 'rain dislikes)) → NIL
```

7.3 Conditional testing

Just as in conventional programming languages, conditional testing is done in Lisp using **if**. The **if** form corresponds to the `if-then-else` construct found in other languages but the terms `then` and `else` are implicit. The call follows the format:

```
(if condition-form then-form else-form)
```

It is possible to omit the `else-form` but if the value of `condition` is `nil` then nothing is done and the returned value of the **`if`** form is `nil`.

Let us now create a function which takes two input radii and compares them to see if they are equal, in which case it would be a circle, otherwise it would be an ellipse. The format of our condition will be:

```
(if (the radii are equal)
  (then it is a circle)
  (else it is an ellipse))
```

Remember that Lisp programs can be indented as much as needed to make them easily readable.

```
(defun shape ()
   (let ((r1 0) (r2 0))
      (terpri)
      (princ "Enter the first radius: ")
      (setf r1 (read))
      (terpri)
      (princ "Enter the second radius: ")
      (setf r2 (read))
      (terpri)
      (if
         (equal r1 r2)
        (princ "This is a circle!")
        (princ "This is an ellipse!"))
      (terpri)))
→ SHAPE
```

If we now call the function and specify that each radius is 3, you should see:

```
(shape)
Enter the first radius: 3
Enter the second radius: 3
This is a circle!

→ NIL
```

The **nil** is returned as the value of the last form evaluated, which is the **terpri**.

There will be many instances when you need to combine a series of expressions for use in programs, especially in conditional forms. This is done quite simply by using **progn**. **defun** and **let** both contain **progn**s in their definitions, thus allowing them to accept one or several forms in a single call.

progn takes a number of forms and evaluates them sequentially from left to right. The last form evaluated produces the value returned by the **progn**.

```
(progn (setf a 'a1) (setf b 'b1) (setf c 'c1))
→ C1
```

when is a derivative of **if** which may prove more useful than **if** in certain circumstances.

The format of a **when** form is:

```
(when condition form(s))
```

`condition` is evaluated first. If it returns `nil`, no `form` is evaluated and the returned value is `nil`. If `condition` returns a non-`nil` value, the `form`s are evaluated sequentially from left to right.

```
(when x a b c) ≡ (if x (progn a b c) nil)
```

unless is similar to **when** in its format but the converse in its mode of operation.

```
(unless condition form(s))
```

`condition` is evaluated first. If the result is non-`nil`, the `form`s are not evaluated and `nil` is the value returned. If the result is `nil` the `form`s are evaluated sequentially from left to right.

```
(unless x a b c) ≡ (if x nil (progn a b c))
```

To illustrate the operation of **when** and **unless**, try replacing the **if** statement in the previous example with the following:

```
(when (equal r1 r2)
     (princ "This is a circle!"))
```

```
(unless (equal r1 r2)
       (princ "This is an ellipse!"))
```

Older versions of Lisp did not have the **if**, **when** or **unless** constructs available, so conditional testing was done using the **cond** form. Just as **car** and **cdr** have been superseded by **first** and **rest** (although they are still available for use), the **cond** form has been superseded by **if**. Whenever possible, **if** should be used in preference to **cond** but you should nevertheless be aware of how to use it. In other languages,

```
(cond (a ...)
      (b ...)
      (c ...)

)
```

would be roughly the same as

```
If a then...
(b ...)
else if b then...
(c ...)
else if c then...
```

cond takes one or more clauses, which are lists of forms. **cond** considers each clause in turn, evaluating the first form in the clause. If that form return true, the remainder of that clause is evaluated and **cond** returns the result (skipping any remaining clauses). If the value of the form is `nil`, **cond** moves to the next clause and evaluates the first form of it.

The format is as follows:

```
(cond (condition1 form(s)1)
      (condition2 form(s)2)
      ...
      (conditionn form(s)n))
```

Each condition is evaluated in turn until a non-`nil` value is found. This terminates any further evaluations of conditions and Lisp then evaluates the corresponding form(s). It is

possible to omit the form(s) of a clause, in which case Lisp returns the value of the condition as the result. If none of the conditions return a non-`nil` value, **`cond`** returns `nil`.

```
(cond (condition1 form11 form12)
      (condition2 form21)
      (condition3 form31))

(if condition1
   (progn form11 form12)
   (if condition2
      form21
      (if condition3
         form31
         nil)))
```

We can now replace the **`when`** and **`unless`** constructs in our **`shape`** function with the following:

```
(cond ((equal r1 r2) (princ "This is a circle!"))
      (t (princ “This is an ellipse!”)))
```

In this instance, we are saying that if *`r1`* and *`r2`* are not **`equal`**, by default it must be an ellipse. This result is forced by placing `t` as our second condition: `t` always evaluates to `t` so if the first test fails, the second will always be true. This is the same as the else construct which is implicit in **`if`**.

[This page intentionally left blank.]

Chapter 8 Iteration and recursion

8.1 Introduction

In most other conventional languages, problems which require repeated execution of the same code are solved using *iteration*, more commonly known as loops. Lisp provides many iterative forms for the construction of loops: it is not possible to include every instance in this introductory text. A full explanation of every iterative construct is given in the discussion under the topic **Control Structure** in Allegro CL Online Help. The more frequently used forms are discussed in this chapter and can be used as a guide to the general operation of these structures.

Many problems can also be solved by a process known as *recursion*. Sometimes you will find yourself in the situation where the function you need to call is actually the function you are writing. We will demonstrate such a problem later. You can overcome this dilemma by making your function recursive. A recursive function is one which calls itself to assist in the solving of a problem. In other words, it is a function which contains a call to itself in its own definition.

8.2 Iteration

Lisp provides a number of constructs for iterative processing. Allegro CL for Windows also makes available an enhancement to Common Lisp: the **`for`** construct. However, the simplest place to start is with the most basic of the iterative structures: **`loop`**. In order to escape from **`loop`**, we will use **`return`** which will be explained shortly.

loop is the simplest of the iterative structures in Lisp. It does not control any variables but continues to execute the code indefinitely. In order to use **loop** effectively, some sort of counter must be implemented to control the number of times the loop is executed.

In an earlier example we defined a function (**meanfour3**) which took in four values and calculated the average. It was tedious enough to write out the prompts and the same code four times; imagine what it would be like if we had to do this ten times. This is an ideal situation for a loop, but we must set up a counter to stop it after four passes through. Note that this is an instance where we have to use **progn** to combine several forms to make a single form for our "else" condition.

```
(defun meanfour4 ()
   (let ((total 0) (count 1))
      (terpri)
      (loop
         (if (equal count 5)
            (return (/ total 4))
            (progn
               (princ "Enter value No.")
               (princ count)
               (princ ": ")
               (setf total (+ total (read)))
               (setf count (+ count 1))
               (terpri))))))
→ MEANFOUR4
```

In this example, we have not only used the counter count to control the loop: we have also used it within the prompt to the user to indicate which number to input. If we run this, we should see:

```
(meanfour4)
Enter value No.1: 2
Enter value No.2: 4
Enter value No.3: 6
Enter value No.4: 8
```

```
→ 5
```

Obviously this works perfectly, but in a more complicated section of code it is easy to forget to update the counter or to position the update wrongly and end up in an infinite loop.

do is another means of executing a section of code repeatedly which also requires that a control variable is set up and modified to terminate the loop. The format of a **do** is shown below:

```
(do ((<parameter1> <initial-value1> <update-form1>)
     (<parameter2> <initial-value2> <update-form2>)
     ...
     (<parametern> <initial-valuen> <update-formn>))
    (<termination test> <zero or more forms> <result-form>)
       <loop body>)
```

Any parameters which are defined at the start of a **do** are bound and assigned values just as if **let** had been used. If the termination test is met, execution of the loop is finished, and the result form is returned as the value of the loop. Otherwise, the forms in the loop body are executed, then the parameters are updated by the update forms and another termination test made. The iteration continues until the termination test is met. Just as in **let**, the initialization and update of variables are done in parallel. In other words it does not matter in what order you place your update list, as on each pass the values are computed and then reassigned. Any reference to the value of a variable in an update is taken as being the value of the last pass.

Let us write a function which calculates the factorial of a number. Control of the loop is through the variable control which is decremented on each pass through the loop. When control reaches zero, the loop is terminated and the answer is returned in the variable result.

```
(defun factorial1 (x)
   (do ((result 1) (control x))
       ((zerop control) result)
      (setf result (* result control))
      (setf control (- control 1))))
→ FACTORIAL1
```

Remembering that **do** executes the initialization and updates in parallel, we could write the function this way:

```
(defun factorial2 (x)
   (do
       ((result 1 (* result control))
        (control x (- control 1)))
       ((zerop control) result)))
→ FACTORIAL2
```

After the first pass, result and control are assigned the values from `(* result control)` and `(- control 1)` respectively.

return is a way of terminating **loop** and **do** forms. As soon as **return** is encountered, execution of the loop is terminated and the value of the form following the **return** is returned as the result. If there is no form following **return**, `nil` is the value returned.

As we saw in our **loop** example, the **return** was given as the `then` clause of the conditional such that when our termination value was achieved, the loop ceased to execute.

for As both **loop** and **do** can be quite cumbersome to use, Allegro CL for Windows includes a **for** construct which simplifies the operation of iterative processes.

```
for ... from ... to ... do ...
```

is an enhancement in Allegro CL for Windows for which there is no equivalent in Common Lisp (although other iterative constructs can be used to achieve the same result). The function **meanfour4** can be modified to use this as follows:

```
(defun meanfour5 ()
   (let ((total 0))
      (for I from 1 to 4 do
         (terpri)
         (princ "Enter value no.")
         (princ I)
         (princ ": ")
         (setf total (+ total (read))))
```

```
      (/ total 4)))
→ MEANFOUR5
```

Once again, the division is the last form evaluated and its value is therefore returned as the value of the function. This is obviously a much cleaner construction than the previous example as there is no need to monitor and update the control variable.

The counter in the **for** loop below is set up to progress in single steps from 0 to 10 but it is possible to override this by including a conditional statement before the **do**. Suppose we wish to perform an iterative task on every second pass through a program, we could use something like this:

```
(for I from 0 to 10
  when (evenp I)
  do (...
```

In this case, the code governed by the DO is only executed when I has an even value.

There are many variations available to the user of the FOR loop including a variety of conditions and forms such as:

```
FOR ... FROM ... TO ... WHILE ... DO
FOR ... FROM ... TO ... UNTIL ... DO
FOR ... FROM ... TO ... UNLESS ... DO
FOR ... IN ... COLLECT
```

A complete explanation of every available form is given under the topic **Control Structure** in Allegro CL Online Help.

8.3 Recursion

Although recursion can be used in many modern languages, it is a technique not as widely employed as it could be. Usually it is obvious which problems are more readily solved by recursion rather than iteration, but (as we shall see shortly) there are occasions when, because of its complexity, a problem can only be solved recursively.

Consider the following function which is a simplified version of the Lisp function **rem**. It expects two positive integers and calculates the remainder when the first (the numerator) is divided by the second (the denominator). Firstly, we must cater for the special cases when

either of the parameters is zero, or the denominator is greater than the numerator. Assuming that none of these apply, the function calls itself and continues to do so until a solution is found. At each call, the value of numerator is (- numerator denominator) for the next stage of the process. When the value of numerator is less than the value of denominator, the value of numerator is the result.

```
(defun my-rem1 (numerator denominator)
   (cond ((< numerator 1) 0)
         ((< denominator 1) 'division-by-zero)
         ((< numerator denominator) numerator)
         (t (my-rem1 (- numerator denominator) denominator))))
→ MY-REM1
```

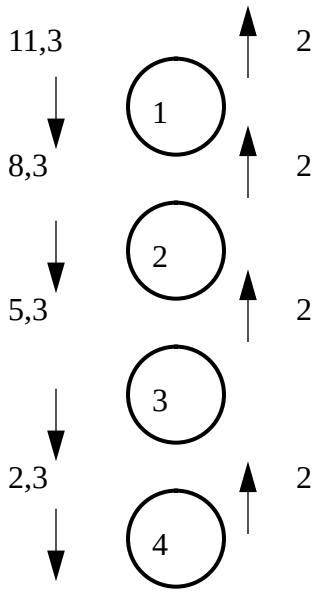
If we now call this function with the arguments 11 and 3, the value of the arguments will be adjusted as follows:

```
Call N Numerator       Denominator
1      11              3
2       8 (- 11 3)     3
3       5 (- 8 3)      3
4       2 (- 5 3)      3
```

At this stage, the numerator is less than the denominator, which terminates the conditional form. The result (2) is held in the numerator and is the value returned by the conditional. You can monitor what happens during this, and any other function, using the **trace** facility. The **trace** facility is turned on using `(trace fn)` and remains active until it is turned off with `(untrace fn)`. To see what happens during **my-rem1**, try:

```
(trace my-rem1)
(my-rem1 11 3)
(untrace my-rem1)
```

What happens during each level of recursion (the flow of control) can be represented in a pictorial manner as in Figure 1 below. Because the flow of control is a straight line, **my-rem1** could have been written using a loop.



Let us now examine a more complicated example: the Fibonacci sequence. Each number in the sequence is the sum of the preceding two:

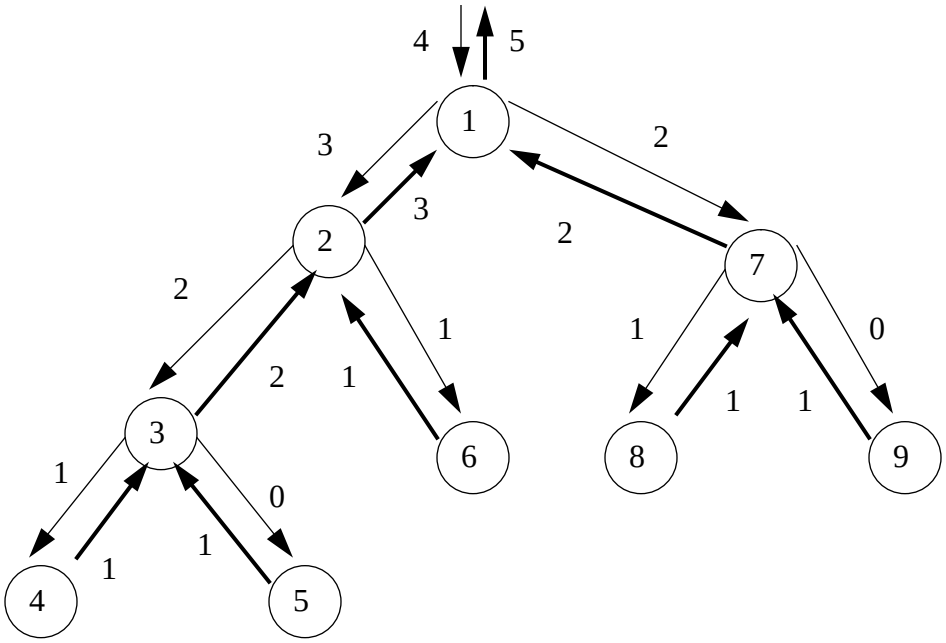
```
1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ....
```

If we wish to calculate a specific term in the sequence, we have to know the preceding two values. This is easily done with a recursive function:

```
(defun fibonacci (a-number)
   (cond ((= a-number 0) 1)
         ((= a-number 1) 1)
         (t (+ (fibonacci (- a-number 1))
               (fibonacci (- a-number 2))))))
→ FIBONACCI
```

This function is said to be doubly recursive because each call to **`fibonacci`** can create two new calls, not one. If we call the function with the argument 4, the diagrammatic representation in Figure 2 below shows what happens. This branching flow of control would

be very difficult to code iteratively but is a natural choice for recursion. Although the complete tree is complicated, at each node there is a simple decision to make. Thus each node can be regarded as a separate problem, and is handled by a single call of **`fibonacci`**. It is this ability to break up large problems into smaller, more easily solved subproblems that makes recursion such a powerful technique.



Both of the examples we have looked at so far involved mathematical calculations but, as with other areas of Lisp, recursion can be applied equally well to manipulate symbols and lists. Consider the following example which will be our own version of the system function **`member`**. By repeatedly examining the first element of the list and calling itself on the rest of the list, the function works out whether an element is present. First of all, we must ensure that the list is not empty. This test will also handle the case when the element is not present and we have run out of elements to test.

```
(defun my-member1 (an-element a-list)
   (cond ((null a-list) 'not-found)
         ((equal an-element (first a-list)) a-list)
         (t (my-member1 an-element (rest a-list)))))
→ MY-MEMBER1
```

Sometimes, an element is hidden in a list within a list. It is not a top- level element. In order to search deeper within a list, we can include a further recursive **cond** clause. Note that our original recursive clause has to be modified slightly.

```
(defun my-member2 (an-element a-list)
   (cond ((null a-list) 'not-found)
         ((equal an-element (first a-list)) a-list)
         ((listp (first a-list)) (my-member2 an-element
                                  (first a-list)))
         (t (my-member2 an-element (rest a-list)))))
→ MY-MEMBER2
```

Neither of these functions is an exact replica of **member**.

[This page intentionally left blank.]

Chapter 9 Data structures

So far, we have used variables and lists for storing our data. Lisp provides other structures for data storage without having enormous lists or a large number of variables. In this chapter we will examine association lists, property lists, arrays and data abstraction.

9.1 Association lists

An *association list* is a list which contains a number of sublists. The first element of each sublist is referred to as the *key.* For example, the sublist:

```
(sports tennis golf swimming)
```

marks tennis, golf and swimming as sports. Retrieving sublists is done with **`assoc`**.

`assoc` The function **`assoc`** takes two arguments, the second of which must be an association list. The first argument is the key and may be any legal Lisp expression. **`assoc`** compares the key with the **`first`** of each sublist in the association list and returns the first sublist whose **`first`** element is **`equal`** to the key list. If the key appears in more than one list, only the first occurrence is "seen". All other occurrences are ignored.

For example, we can define `leisure` as being, amongst other things, `sports`.

```
(setf leisure '((sports tennis golf swimming)
                (sports jogging squash rugby)
                (likes eating sleeping sports)))
→ ((SPORTS TENNIS GOLF SWIMMING)
```

```
    (SPORTS JOGGING SQUASH RUGBY)
    (LIKES EATING SLEEPING SPORTS))
(assoc 'sports leisure)
→ (SPORTS TENNIS GOLF SWIMMING)
```

The other instances of sports are ignored. If the key is not found, **assoc** returns `nil`.

An association list is a type of list and, as such, can be manipulated using **subst**, **first**, **rest**, **cons**, etc.

9.2 Property lists

Property lists are a development of the association list concept. Remember that in Lisp, a symbol can be given a value. That value may be a number, another symbol or a list. In the previous section we learned how to set up an association list by specifying the key as the first item. We can use this to define a person.

```
(setf fred '((familyname smith)
             (sex male)
             (age 30)
             (occupation programmer)))
→ ((FAMILYNAME SMITH) (SEX MALE) (AGE 30)
              (OCCUPATION PROGRAMMER))
```

The symbol `fred` has a number of associated lists as its value. However, Lisp also permits symbols to have properties. A property consists of two elements: a property name and a property value. We can use this concept to define `fred` another way by saying that `fred` has the property names surname, sex, age and occupation and the property values smith, male, 30 and programmer. This will mean that we can ask if `fred` has the property sex and, if so, have the value male returned rather than having to use `(cadr (assoc 'sex fred))` to return male from the association list. The setting up of a property list is slightly more complicated than an association list and we will therefore find out how to retrieve items first.

get is used to return the property value from a property list. It expects two arguments: a symbol name and a property name.

If we assume that we have set up our property list for `fred`, we can retrieve properties as follows:

```
(get 'fred 'sex) → MALE
(get 'fred 'address) → NIL
```

If the property name is not found or if the property value of the name is `nil`, `nil` is returned.

`setf` and **`get`** are used together to create and modify property lists. If a property name is not found, it is added to the property list and the property value assigned. If the property name is found, its value is replaced with the new property value.

Let us look at an example:

```
(setf (get 'fred 'familyname) 'smith) → SMITH
```

The **`get`** returns `nil` and so the property name `familyname` is added to `fred`'s property list and its property value becomes smith.

We can now define the rest of `fred`'s properties thus:

```
(setf (get 'fred 'sex) 'male
      (get 'fred 'age) 30
      (get 'fred 'occupation) 'programmer)
→ PROGRAMMER
```

As many properties as are needed may be added in this way, which means that very exact definitions can be created. This is also a way of updating information in a property list since the value in the **`setf`** form will replace any existing value if the property name is found.

```
(setf (get 'fred 'occupation) 'analyst)→ ANALYST
```

The property value can be any legal Lisp expression. It is possible, for example, to make the value an association list. Suppose that we wish to define a leopard, breaking down the facts into three categories (always, usual and possible). These can be association lists of characteristics which apply under these headings, thus allowing us to create a complex picture of a leopard from which it is relatively simple to retrieve information.

```
(setf (get 'leopard 'always) '((is-a mammal)
                               (blood warm)
                               (eyes forward-facing)
                               (genus felis)))
→ ((IS-A MAMMAL) (BLOOD WARM)
    (EYES FORWARD-FACING)(GENUS FELIS))
(setf (get 'leopard 'usual) '((color yellow-black-spots)
                              (habitat savannah)
                              (prey gazelles)
                              (temperament dangerous)))
→ ((COLOR YELLOW-BLACK-SPOTS) (HABITAT SAVANNAH)
   (PREY GAZELLES) (TEMPERAMENT DANGEROUS))
(setf (get 'leopard 'possible) '((color black)
                                 (habitat zoos)
                                 (prey monkeys)
                                 (temperament tame)))
→ ((COLOR BLACK) (HABITAT ZOOS) (PREY MONKEYS)
   (TEMPERAMENT TAME))
```

In order to retrieve data from this complex structure we must use something like this:

```
(second (assoc 'habitat (get 'leopard 'possible))) → ZOOS
```

Notice that if we had used **`rest`** instead of **`second`**, the returned value would have been `(ZOOS)`.

As you can see, an association list used as the value of a property is an excellent way of saving complicated read-only data. However, accessing individual elements which need to be modified can be a tricky process at this stage, and we would therefore recommend that this structure is not used for any data which requires frequent updating.

`remprop` is used to remove property names and their values from a property list. It takes two arguments: the symbol name and the property name. The property name and its value are removed from the property list of that symbol.

```
(remprop 'fred 'occupation) → T
```

`symbol-plist` displays the contents of a property list. It takes one argument which is the name of the symbol whose property list is

to be displayed. Each property name is immediately followed by its property value.

```
(symbol-plist 'fred)
→ (OCCUPATION ANALYST AGE 30 SEX MALE FAMILYNAME SMITH)
```


9.3 Arrays

All the data structures we have looked at so far involve symbols and lists. However, Lisp does allow the construction of arrays, which are conceptually similar to those in other languages. The most significant difference is that Lisp arrays can accept any type of data: integers, floating point numbers, symbols or even lists. Data types can be mixed in any combination without any need to predefine what will be stored where. An array in Lisp may have any number of dimensions including zero. A zero-dimensioned array will have one element. Different implementations may have different limits but every version should support arrays of up to seven dimensions. The number of dimensions supported can be found in `array-rank-limit`. The numbering of cells in any dimension always begins from zero.

`setf` and

`make-array` are used together to define arrays.

Let us define an array to represent a Tic-Tac-Toe grid.

```
(setf oxo (make-array '(3 3)))
→ #2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
```

This represents the nine squares of the grid. If we were defining a chessboard, we could choose whether we wished to represent the chessmen as simple integers from 1 to 32 or we could generate descriptive symbols. We could even create a property list for each symbol specifying its range of movement and its value in relation to the state of play.

`setf` and

`aref` are used together to retrieve and/or modify cells within an array.

Suppose we wish to position an X in the top right hand corner of the grid, we would do so thus:

```
(setf (aref oxo 0 2) 'x) → X
```

An O can be positioned in the centre of the grid as follows:

```
(setf (aref oxo 1 1) 'o) → O
```

In each case, **aref** locates the cell within the array and **setf** assigns the value.

Used on its own, **aref** returns the value from a cell in an array. For example, we can find out if a cell of the grid has already been used by

```
(aref oxo 1 1) → O
```

9.4 Data abstraction

We can now choose different data structures to suit our requirements when storing information and we have written functions to operate on the data. But what happens if we decide to reformat the data? We could edit every function which uses the data, which would be fine with the small programs we have written so far. However, when we have lots of functions manipulating our data this would be a very tedious task.

The concept of *data abstraction* is to write functions so that they are independent of the data format. Instead, we provide a few functions which access our data directly and ensure that our other functions call these. If we decide to change our data format, which invariably we will, we need only to change these access functions. There should be three types of *access function* associated with the data structure:

1. Functions to build the data.
2. Functions to modify the data.
3. Functions to find items in the data.

These functions may be designed to operate on a specific set of data, or they may be made more general to operate on any data which is held in that format.

Earlier, we created property lists and saw how complicated it can be to retrieve items; we had to remember exactly where and how the data was saved. Let us now write some

functions to help us create, modify and find our data. First of all, we need to be able to create a structure and, for this purpose, we will use our `fred` example. Consider:

```
(defun set-prop (name prop value)
   (setf (get name prop) value))
→ SET-PROP
```

This function can be used to add new properties or to modify existing ones.

```
(defun find-prop (name prop)
   (get name prop))
→ FIND-PROP
```

We now have a way of finding items in our structure. These may seem trivial and not worth the bother. Why not simply use the **`get`** and **`setf`** of **`get`**? Later, if we decide to store properties in an association list, we would have only two small functions to rewrite.

Back to `fred`. We need some functions to access the items in the structure:

```
(defun set-person-familyname (person value)
   (setf (get person 'familyname) value))
→ SET-PERSON-FAMILYNAME
(defun person-familyname (person)
   (get person 'familyname))
→ PERSON-FAMILYNAME
```

We could go on to create these for every item in `fred`'s property list. This is boring. Lisp overcomes the problem of writing all these for you.

`defstruct` short for **`def`**ine **`struct`**ure, **`defstruct`** automatically writes all these functions for us.

Rather than explicitly defining `fred`, we can create a `person` structure into which we can store details of anyone we wish:

```
(defstruct person
   (familyname 'unknown)
   (sex 'unknown)
   (age 'unknown)
```

```
   (occupation 'unknown))
→ PERSON
```

This declares that a `person` is an object with four named components: familyname, sex, age and occupation. When the **`defstruct`** is evaluated, several things happen:

(1) A function called **`make-person`** is generated to create other person data structures with the four components. Thus:

```
(setf fred (make-person))
→ #S(PERSON FAMILYNAME UNKNOWN SEX UNKNOWN
   AGE UNKNOWN OCCUPATION UNKNOWN)
```

will create a new data structure (denoted by #S) with the four components set to the initial values. The values can be specified when the structure is defined, as follows:

```
(setf fred (make-person
              :familyname 'smith
              :sex 'male
              :age 30
              :occupation 'programmer))
→ #S(PERSON FAMILYNAME SMITH SEX MALE AGE 30
   OCCUPATION PROGRAMMER)
```

The component names are preceded by a colon to distinguish them from the component values. If any component is omitted, it is set to the initial value.

(2) The symbol `person` becomes the name of a data type of which other person data structures are elements. This name can be passed to the function **`typep`** which tests whether an object is of a particular type; it returns non-`nil` if it is, `nil` if it isn't.

```
(typep fred 'person)
→ (#<STRUCTURE-CLASS PERSON #X38835C>
    #<STANDARD-CLASS STRUCTURE-OBJECT #XD83B8>
    #<BUILT-IN-CLASS T #XD6AF8>)
```

(3) A function called **`person-p`** is defined which also returns non-`nil` if its argument is an object of the type `person`, and `nil` if it isn't.

```
(person-p fred)
→ (#<STRUCTURE-CLASS PERSON #X38835C>
      [...])
```

(4) The four components become functions, each with one argument, that return the values held in each.

```
(person-familyname fred) → SMITH
```

(5) The data can be modified using **`setf`** and the appropriate selector using this construction:

```
(setf (person-component name) value)
```

where

`component` is the selector

`name` is the name of the structure

`value` is the new value

To modify `fred`'s occupation, we would write:

```
(setf (person-occupation fred) 'analyst) → ANALYST
```

All other components can be modified in this way.

Finally, the `#s` syntax can be used to read in the structure:

```
(person-p '#s(person familyname fred))
→ (#<STRUCTURE-CLASS PERSON #X38835C>
    #<STANDARD-CLASS STRUCTURE-OBJECT #XD83B8>
    #<BUILT-IN-CLASS T #XD6AF8>)
```

[This page intentionally left blank.]

Chapter 10 Lambda

Throughout this document we have emphasized the important role of functions in Lisp programming and we have defined many functions using **defun**. There will be times, however, when a function will be called on so few occasions that you may consider it a waste of time to create one. In Lisp, it is possible to establish a "temporary" function which is "discarded" once a value has been returned. This is done by `lambda` and is known as a lambda expression. It differs from a function defined using **defun** in that it does not require a name since it is not called as a built-in function.

In this chapter you will notice the appearance of the character sequence `#'`. This indicates that the elements following are code. In the same way that `'(form)` is equivalent to `(quote (form))`, `#'(form)` is equivalent to `(function (form))`.[1]

For example,

```
#'(lambda (x)(* x x x))
```

is a function of the argument which returns the cube of its argument. But how would we use it, since it has no name? The answer is that there are functions which expect functions as their arguments. We can either pass them a function name or a lambda expression. Let us now see lambda expressions in action.

10.1 mapcar

mapcar is known as a mapping function since it "maps" the operation of a function over the items in a list and collects the results as a list. If we look at some examples, the operation should become clear:

1. It is legal to precede a lambda expression with the single quote (it will eventually achieve the desired result) but it is much more efficient to use `#'`.

```
(mapcar #'evenp '(1 2 3 4)) → (NIL T NIL T)
```

In this instance, the function **evenp** is applied in turn to each item in the list and **mapcar** returns a list of the values returned. Let us now create a list of the cubes of these numbers. We have no built-in function to do this, but we do not want to define a named function just for this task: however, we can use the lambda expression we saw in the previous section.

```
(mapcar #'(lambda (x) (* x x x)) '(1 2 3 4))
→ (1 8 27 64)
```

mapcar can be used with functions of more than one argument:

```
(mapcar #'equal '(1 2 3) '(1 4 9 16)) → (T NIL NIL)
```

equal requires two arguments for comparison. Here it is applied three times comparing 1 with 1, 2 with 4, and 3 with 9. **mapcar** continues until the end of the shortest list is reached, ignoring any remaining elements.

```
(mapcar #'max '(1 2 3 4) '(1 4 9 16) '(11 2 1 0))
→ (11 4 9 16)
```

10.2 apply

There will be situations where you wish to use the elements of a list (such as the result of **mapcar**) as arguments to a function, but that function cannot operate on a list. For example, if we wish to add the results from the previous example, the addition function will not accept a list as its arguments; it needs to know the individual elements to add together. The function **apply** extracts the elements from the list and provides them as arguments for the function:

```
(apply #'+ '(11 4 9 16)) → 40
```

Of course, we can combine these two operations:

```
(apply #'+ (mapcar #'max '(1 2 3 4)
                   '(1 4 9 16)
                   '(11 2 1 0)))
→ 40
```

You will find **apply** a very useful function, not just in conjunction with **mapcar**.

10.3 Filtering

Filtering is the process of looking at lists and deciding whether or not to discard elements. This is another area where the ability to pass functions as arguments is crucial. Two built-in functions, **remove-if** and **remove-if-not**, are provided to help with this process.

Both functions expect a predicate of one argument and a list. They operate in a similar way to **mapcar**; each element of the list is passed to the predicate in turn for testing.

remove-if returns a list containing any elements which returned `nil` from the test:

```
(remove-if #'(lambda (x) (< x 4))
   '(1 2 3 4 5 4 3 2 1))
→ (4 5 4)
```

remove-if-not is the converse of **remove-if**; it returns a list containing any elements which returned a non-`nil` value from the test:

```
(remove-if-not #'(lambda (x) (< x 4))
   '(1 2 3 4 5 4 3 2 1))
→ (1 2 3 3 2 1)
```

In the above example, **remove-if-not** expected a function of one argument and we wanted to give **<** two arguments; `lambda` solved the problem. The second argument to **<** was a constant, but it could equally well have been a variable.

```
(defun cut-off (value a-list)
   ; returns a list of
   ;elements which exceed
   ;value
   (remove-if-not
      #'(lambda (x)
           (< x value))
      a-list))

(cut-off 4 '(1 2 3 4 5 4 3 2 1))
```

```
→ (1 2 3 3 2 1)
(cut-off 2 '(1 2 3 4 5 4 3 2 1))
→ (1 1)
```

10.4 Functions as arguments

If we want to write a function which accepts a function as an argument, we need a way of telling Lisp to use the argument as a function. **`funcall`** does this.

`funcall` can take several arguments. The first must evaluate to a function which is applied to the remaining arguments.

```
(defun trace-call (fn argument)
   (print fn)
   (princ " called on ")
   (prin1 argument)
   (print fn)
   (princ " returns ")
   (prin1 (funcall fn argument)))
→ TRACE-CALL
(trace-call #'first '(a b))
→ FIRST called on (A B)
  FIRST returns A
  A
```

This gives a hint as to how **`trace`** is implemented.

`(funcall fn argument)` is not the same as `(fn argument)`. Remember that Lisp assumes that the first item in a list is the name of a function. `(fn argument)` would cause Lisp to look for a function called **`fn`**. `(funcall fn argument)` causes **`fn`** to be evaluated as a variable to produce the returned function to be called (in this case, **`first`**).

```
(defun do-to-number (number minus-fn plus-fn)
   ;if number is negative minus-fn is applied to it
   ;otherwise plus-fn is applied to it
   (funcall
      (if (< number 0)
```

```
         minus-fn
         plus-fn)
      number))
→ DO-TO-NUMBER
(do-to-number -1 #'- #'(lambda (x) (* x x x)))
→ 1

(do-to-number 4 #'- #'(lambda (x) (* x x x)))
→ 64
```

Here is a simple version of **mapcar**, which can only operate on one list:

```
(defun my-mapcar (fn a-list)
   (if (null a-list)
      nil
      (cons
        (funcall fn (first a-list))
        (my-mapcar fn (rest a-list)))))
→ MY-MAPCAR
```


10.5 Optional arguments

As their name implies, optional arguments may or may not be included in a call to a function which includes them in its argument list. They are used in general-purpose functions which cannot know how many arguments to expect, such as **+**, **-**, *****, **/** and **list**. We can incorporate them in our own functions when we wish a default value to be applied unless told otherwise.

If we now look at a simple example we can see how they work. The **log** function in Lisp defaults to the base *e* unless another base is specified. Let us suppose it only works to the base *e* but we need to be able to work to any base. We can write our own function to do this:

```
(defun my-log1 (num base)
   (/ (log num) (log base)))
→ MY-LOG1
```

Every time we call this function we must specify the number and the base:

```
(my-log1 8 2) → 3.0
```

Now let us suppose that we want to default to base 10 if no base is given. This can be done by specifying base as an optional argument: you may or may not include it, as you wish.

`&optional`	indicates that the argument following is optional.

Using `&optional`, we can modify our function to default to base 10 if no base is given.

```
(defun my-log2 (num &optional base)
   (if (not base)
      (setf base 10))
      (/ (log num) (log base)))
→ MY-LOG2
```

We now have the option of specifying a base if we wish, or defaulting to base 10 if we don't.

```
(my-log2 8 2) → 3.0
(my-log2 100) → 2.0
```

There is, however, another way of specifying a default parameter, by including it in the argument list.

```
(defun my-log3 (num &optional (base 10))
   (/ (log num) (log base)))
→ MY-LOG3
```

This is a much cleaner construct than **`my-log2`**, which works in exactly the same way:

```
(my-log3 8 2) → 3.0
(my-log3 100) → 2.0
```

There may be any number of optional arguments.

`&rest`	Works in a similar way to `&optional`, except that it takes a single argument whose value becomes a list of all the arguments given, other than required and optional arguments.

In Lisp, there is no function to give the sum of squares, so let us write one:

```
(defun sum-of-squares1 (a b)
   (+ (* a a) (* b b)))
→ SUM-OF-SQUARES1
```

This will give us the sum of the squares of any two arguments we supply:

```
(sum-of-squares1 2 4) → 20
```

But how do we write it so that it takes any number of arguments? We use `&rest`.

```
(defun sum-of-squares2 (&rest nums)
   (let ((result 0))
      (for i in nums
         do
         (setf result (+ (* i i) result)))
      result))
→ SUM-OF-SQUARES2
```

Any arguments supplied are made into a list. On every pass through the **`for`** loop, each number is squared and added to result.

```
(sum-of-squares2 1 2 3 4) → 30
```

[This page intentionally left blank.]

Chapter 11 Macros

The concept of macros in Lisp is similar to that in other languages: they allow the user to use a shorthand for sections of code that are used repeatedly. Let us say, for example, that we are adding items to lots of lists. We will repeatedly have to write:

```
(setf variable (cons value variable))
```

It would be nicer and cleaner if we could simply write something like:

```
(push value variable)
```


Why not define **push** as a function? Consider `(push 'x y)`. Before the function call, y would be evaluated and its value given to **push**. **push** would not know about y and therefore would not be able to store the new list into it. The solution is to make `(push 'x y)` rewrite itself as `(setf y (cons 'x y))`. This can be achieved by defining **push** as a macro, not as a function.

When a Lisp form is being evaluated or compiled, if it is a macro call, the macro is called to compute the actual form to be evaluated or compiled. The macro receives parts of the macro call unevaluated. The actual form may itself be a macro call in which case the process is repeated. This process of converting a macro call into the actual form for evaluation is known as macro expansion.

During compilation, the macro calls are expanded and lost, which makes the code more difficult to debug (for example, macro calls may not be traced). We would therefore recommend that whenever possible, a section of code should be defined as a function rather than a macro. **defun** and **let** are actually macros rather than functions, which is why they do not evaluate their arguments in the way that **+**, **mapcar**, **cons** and all the other functions do. A macro is defined using **defmacro**, which is similar to **defun**.

defmacro requires the same information as **defun**: the name of the macro, the arguments to it and the task the macro is to perform.

push is already defined in Lisp, but let us suppose that it does not exist. We could define it very simply with something like this:

```
(defmacro my-push (value variable)
   (list 'setf variable
      (list 'cons value variable]
→ MY-PUSH
```

To see if a macro has been correctly defined, we can expand the call with **macroexpand-1**.

macroexpand-1 expects a macro call as its argument. It expands the macro call once and returns two values: the expansion and `t`. If the argument is not a macro call, it returns the argument and `nil`. Allegro CL for Windows makes the facility available from a menu.

```
(macroexpand-1 '(my-push 'x y))
→ (SETF Y (CONS 'X Y))
  T
```

The definition of a macro can be greatly simplified by using **backquote**. **backquote** is not specifically related to macros but, since it can play such an important part in macro definitions, this is an appropriate time to introduce it.

backquote is a special form of **quote** which tells Lisp not to evaluate what follows except those objects preceded by a comma.

For example:

```
(setf name 'test)
→ TEST
'(this is a ,name list)
→ (THIS IS A TEST LIST)
```

However, this may not always produce the result you require when working with lists:

```
(setf name '(list within a))
→ (LIST WITHIN A)
`(this is a ,name list)
→ (THIS IS A (LIST WITHIN A) LIST)
```

If you do not wish `(list within a)` to be enclosed in parentheses, you could use **`append`** or the special construct `,@` within **`backquote`**.

```
`(this is a ,@name list)
→ (THIS IS A LIST WITHIN A LIST)
```

When writing macros, the backquote allows the macro to be thought of as a template with the expressions filled in as required. Suppose, for example, that we wish to write our own version of **`let`**. First of all we have to create a couple of helping functions:

```
(defun vars-of (vars+values)
   (for var-value-pair in vars+values
      collect
      (if (symbolp var-value-pair)
         var-value-pair
         (first var-value-pair]
→ VARS-OF
(defun values-of (vars+values)
   (for var-value-pair in vars+values
      collect
      (if (symbolp var-value-pair)
         nil
         (second var-value-pair]
→ VALUES-OF
```

Now we can define our **`let`**:

```
(defmacro my-let (vars+values &rest forms)
   (cons
     (cons 'lambda
       (cons (vars-of vars+values) forms))
     (values-of vars+values)))
→ MY-LET
```

Using the backquote, this can be simplified thus:

```
(defmacro my-let (vars+values &rest forms)
   `((lambda ,(vars-of vars+values) ,@forms)
     ,@(values-of vars+values]
```

gensym When writing macros you may wish to create new symbols which you require to be unique to that code. This is done using **gensym** which is short for '**gen**erate **sym**bol'. **gensym** allows you to create new symbols that are not normally seen by the user and are distinct from any with the same name that may be defined outside that area of code. Although **gensym** is not specifically allied to macro definition, its use when writing macros is obvious.

Chapter 12 List storage

This chapter explains how lists are stored and modified in memory and thus allows us to see what happens with some of the functions we have discussed. We will also look at some of the more complicated functions.

12.1 How lists are stored

Lisp uses a method of linking addresses to store lists in memory. Each element of a list is represented by two pointers in a single address. These pairs of pointers are called cons cells because as we will see later, they are created by the function **cons**. If we set up an example containing only top-level elements, this should become clear:

```
(setf a-list '(this is an ordinary list))
→ (THIS IS AN ORDINARY LIST)
```

The memory locations of this list would look something like this:

```
1271:       8022        1273
1272:       8551        1275
1273:       9644        1272
1274:       9640        0
1275:       7998        1274
```

The numbers in the first column represent the addresses within memory at which the pointers are stored. Those in the second column represent the actual addresses where the elements are stored. The third column contains the linking addresses of the pointers for the next element.

If we start at the first line, the first pointer (the *car*) indicates that the element is stored at address 8022 while the second element (the *cdr*) tells us that the pointers for the next ele-

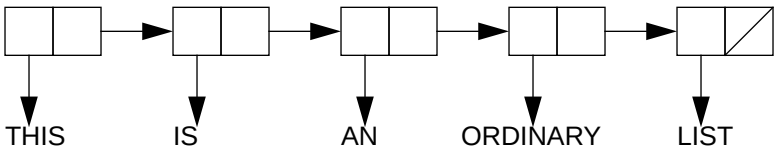
ment are to be found at 1273. At 1273, the pointers indicates that the element is stored at 9644 while the pointers for the next element are to be found at 1272.

Following this process through, we eventually end up at 1274 which says that the fifth element is stored at 9640. The 0, signifying that there is no more linking information, indicates that this is the end of the list (`nil`).

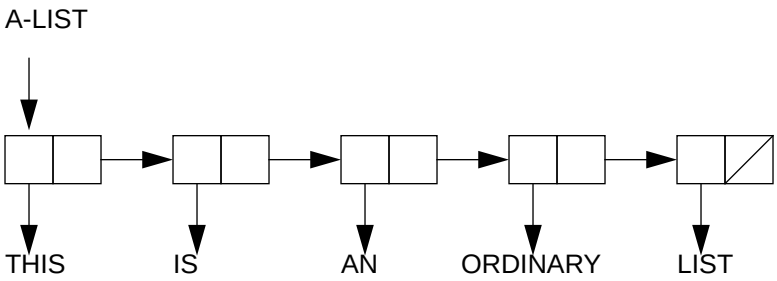
There are two things to note about this process:

1. The order of the elements in a list is determined by pointers and not by the positions where they are stored in memory.
2. The pointers are stored in a different area of memory from the elements themselves.

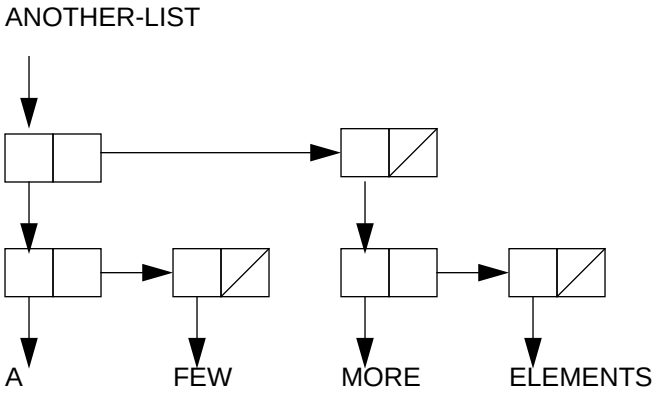
For most of this section, it is not necessary to go into such detail and we will therefore use pictorial "box-and-arrow" representations for the examples. The "box-and-arrow" representation of the above example list is as follows:



Each double box represents a pair of pointers. The vertical arrows show the pointers to the elements, while the horizontal arrows represent the linking pointers to the next elements. The diagonal bar in the right-hand box signifies the end of the list; i.e. there is no pointer. From our example, this diagram is incomplete as it does not show the name of the list. The next figure shows how the name is directed to its value:

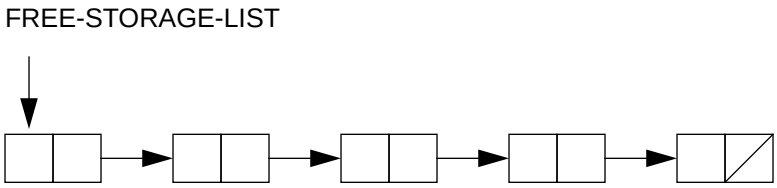


Any list structure can be demonstrated with this notation. Assuming that we have another list named `another-list` which has the value `((a few) (more elements))`, we can see from the next diagram that the first pointers direct us to another pair of pointers which point to the actual elements:



As long as the memory is not full, there are always addresses set aside to store pointers. These are known collectively as the *free-storage-list*. The first pointer at each of these addresses indicates that these pointers have not been used, while the second points to the

next unused address. In this way, all these unused addresses are linked together in a single list:



12.2 How lists are modified

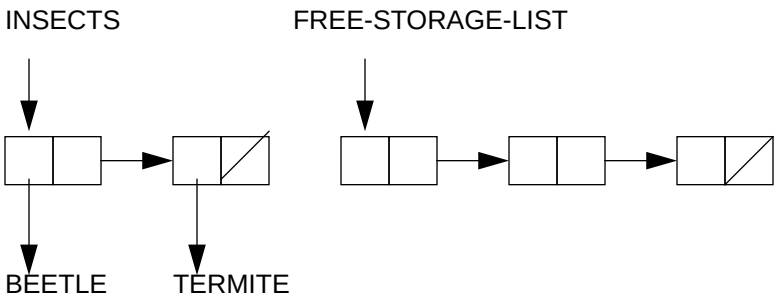
We can now look at the way functions affect memory structures. Some of these functions have been demonstrated in earlier chapters; the others are introduced here for the first time.

cons adds top-level elements to the front of a list.

Let us look at an example:

```
(setf insects '(beetle termite))
```

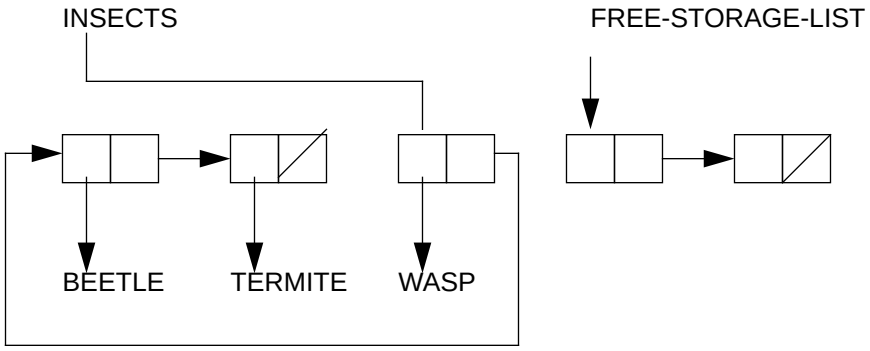
The list `insects` now has the value `(beetle termite)`:



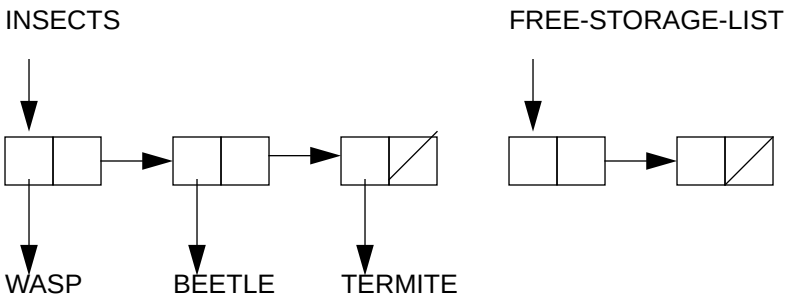
If we now do:

```
(setf insects (cons 'wasp insects))
```

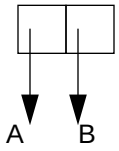
the **cons** takes the first address from the free-storage-list and uses it to point to the symbol `wasp`. The second of the pair of pointers points to the address of the first element in the list `insects`. `insects` is modified to point to `wasp`'s pointers, while free-storage-list is in turn modified to point to the next unused address:



The diagram of these modifications can be simplified thus:



It should be noted that in special circumstances **cons** can have two symbols for its arguments. In this instance, each symbol uses only one of the pointers. For example, `(cons 'a 'b)` is shown thus:

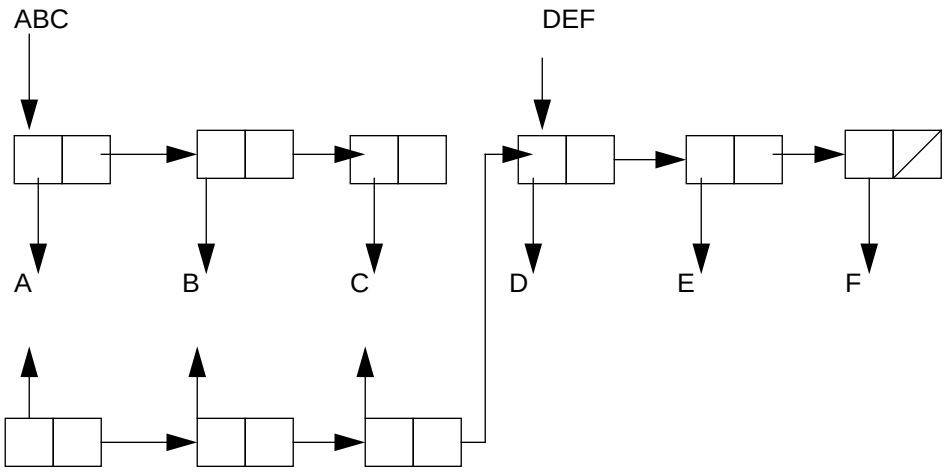


This structure is printed as (A . B) which is known as a dotted pair.

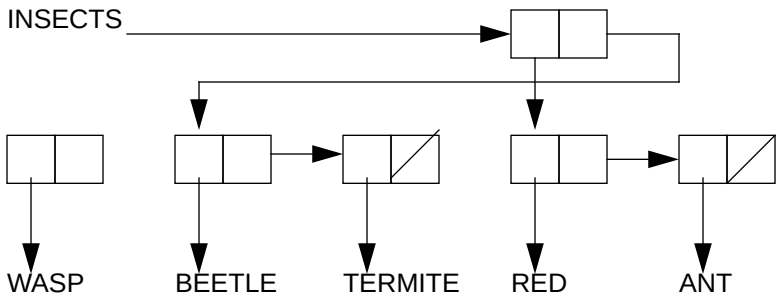
append creates a new list by merging the elements in its arguments.

Assuming that `abc` has the value `(a b c)` and `def` has the value `(d e f)`, these can be joined to produce `abcdef`. However, it does not affect the original lists but makes a copy of the first list with the final pointer modified to point to the next list. This copy uses addresses from free-storage-list:

```
(setf abcdef (append abc def))
→ (A B C D E F)
```



`rplaca` expects two arguments: the first must be a list but the second may be a symbol or a list. It replaces the first element of the list with the second argument. To do this, it changes the first pointer in the first argument to point to the new element. **`rplaca`** is mnemonic for ‘replace CAR’.



```
(rplaca insects '(red ant))
→ ((RED ANT) BEETLE TERMITE)
```

rplacd expects two arguments: the first must be a list but the second may be a symbol or a list. It replaces the **rest** of the first argument with the second argument. **rplacd** is mnemonic for 'replace CDR'.

Assuming the list `computer-fact` has the value `(computers are useful)`:

```
(rplacd computer-fact '(can be very expensive))
→ (COMPUTERS CAN BE VERY EXPENSIVE)
```

nconc joins lists together. It expects two or more lists as its arguments.

Assuming that ABC has the value (A B C) and DEF has the value (D E F), these can be joined to produce ABCDEF.

```
(setf abcdef (nconc abc def))
→ (A B C D E F)
```

The last list, in this case DEF, remains intact.

remove expects two arguments; the first must be a symbol and the second a list. It copies the list and removes each top-level occurrence of the symbol. It does not affect the original list.

Assuming that `votes` has the value `(yes yes no no yes no)`:

```
(remove 'yes votes) → (NO NO NO)
```

delete is the destructive version of **remove**. It does destroy the original list and should therefore be used with caution.

When modifying lists with the above functions, two things should be borne in mind:

1. **nconc**, **rplaca**, **rplacd** and **delete** operate on the original lists and are therefore very efficient, although destructive.
2. **append** and cons are much safer, but they are not as efficient since they use up addresses in the **free-storage-list**.

12.3 Garbage collection

Obviously, continued use of functions such as **append** and **cons** could result in all the free space being used up, but you do not have to worry about this as it is taken care of automatically. Unlike many other languages, Lisp actually monitors the free-storage-list and, as addresses become used up, it triggers a process known as garbage collection. This process reclaims these addresses once they are no longer being used. When the garbage collector is in action you will see an indicator appear on the screen. The indicator will change shape to reflect different operations. It is also possible for you to trigger a garbage collection by typing `(gc)`, although this should not be necessary at this level.

12.4 equal, eql and =

As well as **equal** and = which we met earlier, there is a third predicate available for testing equality: **eql**. **eql** is not the same as **equal**, as we will see. First of all, let us define three lists and see how they are stored in memory:

```
(setf fruit-one '(apple orange pear))
→ (APPLE ORANGE PEAR)
(setf fruit-two '(apple orange pear))
→ (APPLE ORANGE PEAR)
(setf fruit-three fruit-two)
→ (APPLE ORANGE PEAR)
```

Although these appear to be the same, when we look at how they are stored, we see that the first form creates three pairs of pointers which are used to find the value of `fruit-one`, while the second form creates a different set of pointers to point to the value of `fruit-two`. Even though these have the same value, they are assumed to be different because they were defined separately. The third form uses `fruit-two` as its value and so the same three pairs of pointers are used to find the value of `fruit-three`.

equal does not consider the address pointers when looking for equality and therefore assumes that these three lists are equal because they have the same value.

eql, on the other hand, is more precise: it only considers objects to be the same if they share the same pointers. To **eql**, `fruit-two` and `fruit-three` are equal, whereas

`fruit-one` is not. **`eql`** does not examine the values and, as a result, is much faster, especially if used with very large lists.

Care must be taken when operating on lists that are **`eql`**, as any changes affect them all. For example, removing any element from `fruit-two` also affects `fruit-three`, which could be an unexpected side effect.

Where does the predicate **`=`** fit in? Obviously, this is the best format for mathematical testing although it is possible to used **`equal`** or **`eql`**. However, **`=`** only operates on numbers and will generate an error if its arguments are not all numbers. Also, it converts numbers to the same type before testing the values in the same way as **`equal`**. **`equal`** does no conversion and considers numbers of different types to be unequal even if they have the same numerical value:

```
(equal 1 1.0) → NIL
(= 1 1.0) → T
```

List searching functions (e.g. **`member`** and **`subst`**) use **`eql`** for testing equality. If in a certain situation the use of **`equal`** or **`=`** would be preferable, it is possible to specify the equality test for a particular call. Thus:

```
(member 'an-element any-list :test #'equal)
```

In this example, **`equal`** is used for equality testing during evaluation of this form. As soon as evaluation is complete, the default predicate (**`eql`**) is restored.

This principle can be applied to other searching functions. Further details can be found under the headings **Lists** and **Sequences** in Allegro CL Online Help.

Chapter 13 An introduction to CLOS

In this chapter, we will give a brief introduction to CLOS, the Common Lisp Object system. We will develop an extended example, showing how to use CLOS to construct a (toy) object oriented database for tracking inventory in a general store. The focus will be on the two fundamental notions that extend Common Lisp into CLOS: the *class* and the *generic function*.

13.1 Generic functions

Generic functions are functions which behave differently depending on the type and identity of the arguments which are passed to them. This is, of course, not a very strong statement, since most functions will have different side effects and return values, depending on whether they are passed the number 2 or the list (1 2 3 4). Generic functions, however, provide a way of codifying these different behaviors without writing code which is a mass of **cond**, **case** and **typecase** forms. Suppose, for the sake of a rather foolish example, we want to write a function which takes one argument, *x*, and returns `(car x)` if *x* is a list and returns *x* otherwise. Without generic functions, we would have to do something like this:

```
(defun stuff (x) (if (listp x) (car x) x))
```

So far so good. But say we had other special cases, say if *x* happened to be a lexical closure, and yet another if it was a symbol. We would have to (a) have access to the source code for **stuff** and (b) be able to hack it apart and reconstruct it with enough **cond** and

case (or related) forms to make sure it had the desired behavior. A source code management and debugging nightmare would ensue after a very small number of such iterations.

The generic function approach to the same problem is simpler to write and results in less complex, more modular code. The above function **stuff** would be rewritten as follows:

```
(defgeneric clos-stuff (x))
(defmethod clos-stuff ((x list)) (car x))
(defmethod clos-stuff (x) x)
```

The first form announces a new generic function, called **clos-stuff**, which has a lambda list `(x)`. The next two forms describe the behavior of this function in two cases, if the argument *x* is a list, and if it is not. These two possible behaviors, called *methods*, are "specialized" versions of the generic function in that they are applied only in specific situations. The work of deciding what case we are in, which was done explicitly in the **if** statement in **stuff**, is done implicitly using the generic function's built-in discrimination between different types of arguments.

Why is this an advance? Well, suppose we want to add another possibility: if *x* is a symbol, return its symbol-function. We don't need to hack at the code for **clos-stuff**, we don't even need to see it, all we need to know is the lambda list and then we can do the following:

```
(defmethod clos-stuff ((x symbol)) (symbol-function x))
```

We have simply added a method to the generic function **clos-stuff** to cover the case when the argument is a symbol.

This is admittedly a foolish example, but the power of the machinery is evident.

13.2 Classes

If all that we could do with generic functions is to distinguish between symbols and lists, they would not be very useful. The power of CLOS is that it allows the programmer to define entirely new classes of Lisp objects which a generic function can distinguish from one another. If, for instance, we wanted to build a Lisp system capable of drawing graphics on a screen or writing the corresponding images into a PostScript file, we could define a class of screens and a class of PostScript files and define a **draw** generic function which

would have methods for each of the different classes. If we later decided to allow bitmap files rather than PostScript files, we would simply define the appropriate classes and methods.

Let us give an example of this, continuing our **clos-stuff** example. (We will be in a position to construct a more sensible example shortly.) We first define a new class named `bell` and then we add a method specifying how **clos-stuff** should act on members of the class `bell`:

```
(defclass bell () ())
(defmethod clos-stuff ((x bell)) "Ding dong.")
```

The syntax of **defmethod** now becomes clearer: the **cadr** of each element of the specialized lambda list (`bell` is the **cadr** of `(x bell)`) is the name of a class. In fact, `list`, `symbol` and `t` are names of CLOS classes which correspond to Common Lisp types, but any CLOS class will do. Hence,

```
(clos-stuff (make-instance 'bell)) → "Ding dong."
```

A class is essentially a collection of Lisp objects that have something in common. As we have seen, one thing they have in common is the way in which a generic function will treat them. Classes can be used to dictate that a generic function will always behave a particular way when presented with a particular kind of object.

Two concepts which give classes power are *slots* and *inheritance*. A slot in a class is just like a slot in a structure: it is a named spot where information can be stored. Inheritance allows the programmer to make incremental changes, by saying that one class is similar to another or includes structure and behavior from another. We will explore inheritance and slots in more detail in the next two sections.

13.3 Slots and a better example

It is time for a more sophisticated example to give some concreteness to what we have been discussing. Suppose we are trying to track inventory at a general store. There will be all kinds of things to keep track of: books, shaving cream, food, etc. So let us define a class to take care of all of these, called `merchandise`.

Each instance of the class `merchandise` will correspond to a particular kind of thing our store sells. So there will be an instance for Three Pronged Electric Widgets, one for copies of *Murder on the Orient Express* (the book) and another for *Murder on the Orient Express* (the video). Each such instance will have several properties: an identifying name, the quantity in stock, the cost at which they can be acquired, and the price at which they are sold.

```
(defclass merchandise () (name cost price quantity))
```

You create an instance of a class with **`make-instance`**. So, if we now evaluate

```
(setq item (make-instance 'merchandise))
```

we will be given a fresh instance of this class. What are the values for the various slots we have defined? Since no value was given, these slots are unbound, just like an uninitialized variable. We can use **`setf`** to assign values to the slots:

```
(setf (slot-value item 'name) "Three Pronged Widgets")
```

and so on. But this gets tedious. The easier (and less error-prone) technique for accomplishing this is to define the class in such a way that initialization of the various slots is possible at the time that an instance is created:

```
(defclass merchandise ()
    ((name :accessor name :initarg :name)
     (price :initarg :price :accessor price)
     (cost :initarg :cost :accessor cost)
     (quantity :initarg quantity :initform 0 :accessor quantity)))
```

This form defines the class `merchandise` and specifies the four slots of the class: `name`, `price`, `cost`, and `quantity`, just as was done with the simpler **`defclass`** form above. But now, we have further defined *initargs* (initialization arguments) and *accessors* for each slot and an *initform* for the `quantity` slot. The `:initarg` value names the keyword argument that can be used in a **`make-instance`** form to specify the slot value for an instance of the merchandise class. The `:accessor` names the function that can be used to access the slot value. The `:initform` specifies the form to be evaluated to get the value of a slot if no value is specified in the **`make-instance`** form.

We can now construct an instance of `merchandise` using initargs:

```
(setq item (make-instance 'merchandise
             :name "Three Pronged Widgets"
             :price 5.69
             :cost 1.39))
```

This time `item` comes equipped with a name, a price and a cost, all specified in the **`make-instance`** form. We can access these slot values easily enough using **`slot-value`**:

```
(slot-value item 'name) → "Three Pronged Widgets"
```

or we can use the accessors defined in the **`defclass`** form above:

```
(name item) → "Three Pronged Widgets"
```

The `:initarg :name` part of the **`defclass`** form instructs CLOS that `:name` is to be used as a keyword to indicate the name of an instance of `merchandise`. The `:accessor` name part instructs CLOS to define a (generic) function called **`name`**, which can be used to return the value of the name slot of an instance. We can also use the accessor and **`setf`** to change a slot value:

```
(setf (name item) "Two Handled Pitchforks")
→ "Two Handled Pitchforks"
```

What about `quantity`? We didn't specify a value for this slot yet...

```
(quantity item) → 0
```

The `:initform 0` part of the **`defclass`** form instructs CLOS that, in the absence of an explicit `:quantity` in a `(make-instance 'merchandise [...])` form, it should construct an instance with `quantity` set to 0.

Now let's add some bookkeeping functions to help us manage our inventory.

```
(defgeneric sell (item))
(defgeneric buy (item qty))
(defvar *cash-on-hand* 1000)

(defmethod sell ((item merchandise))
   (if (< (quantity item) 1)
      (error "~&~S is sold out. Sorry.~%" (name item))
```

```
      (progn
        (decf (quantity item))
        (incf *cash-on-hand* (price item)) item)))

(defmethod buy ((item merchandise) qty)
   (decf *cash-on-hand* (* qty (cost item)))
   (incf (quantity item) qty) item)
```

A few points should be made. As before, the `merchandise` in the lambda list of these two methods is instructing CLOS that these methods are to be used when item is an instance of the class `merchandise`. Notice that `qty` does not have any such "specialization" attached to it: any class of `qty` will match this method. In reality, one would always specify a number for `qty` or else an error would be signaled later (when we try to multiply it by the cost of the item) but CLOS need not be told everything, just enough so it can choose which method is appropriate in a situation.

We can now buy and sell some items:

```
(buy item 5)
(sell item)
(sell item)
```

and we will be left with three pitchforks and $1004.43.

13.4 Class inheritance

So far it may be hard to see what the big deal is. Everything we have done could have been accomplished with a lot less fanfare using **`defstruct`** and some ordinary (non-generic) functions. Now we will begin to see the power of CLOS however, using *class inheritance*.

Merchandise itself falls naturally into different classes: food, books, sundries... From the point of view of our inventory control scheme, one natural division is between the taxable and the nontaxable. A taxable item is just like a nontaxable one except that the store owner is responsible for collecting a sales tax on the item and paying it over to the state. So let’s define a class of taxable merchandise:

```
(defclass taxable-merchandise (merchandise) ())
```

This class inherits from the class `merchandise`, meaning that it has all the same slots (name, quantity, etc.) and, furthermore, that every method that applies to `merchandise` can also apply to `taxable-merchandise`. Thus, say you try to apply a particular generic function, say the generic function **`sell`** defined above, to some instance of `taxable-merchandise`. Even though no method for **`sell`** for an argument of class `taxable-merchandise` has been defined, CLOS checks to see if there is a method for any class that `taxable-merchandise` inherits from (i.e. for any *superclass* of `taxable-merchandise`). Since there is a method for `merchandise`, that method is applied.

So `taxable-merchandise` is a subclass of `merchandise` and inherits all of its slots and methods. This is almost what we want, but not quite. Every time we sell something taxable, we must collect a little extra and we also incur a liability with the state. What we need is a different method for **`sell`** for `taxable-merchandise`.

```
(defconstant *sales-tax-rate* .08)
(defvar *sales-tax-owed* 0)
(defmethod sell ((item taxable-merchandise))
   (if (< (quantity item) 1)
      (error "~&~S is sold out. Sorry.~%" (name item))
      (progn
         (decf (quantity item))
         (incf *cash-on-hand* (price item))
         (incf *cash-on-hand* (* *sales-tax-rate* (price item)))
         (incf *sales-tax-owed*
            (* *sales-tax-rate* (price item))) item)))
```

Now we will keep track of what we owe the state and make sure to collect the right amount from each consumer. When we call **`sell`** on something taxable (i.e. an instance of the class `taxable-merchandise`), it will use the method just defined instead of the one inherited from `merchandise`. On the other hand, **`buy`** will continue to find the inherited method, since we haven't defined a new one. The point is that when you call a generic function on some Lisp object, CLOS first checks to see which methods could apply, either directly or by inheritance from some superclass of the object. It then chooses whichever method is most closely designed for the particular object at hand, by choosing a method that was defined for its class if possible. (If that is not possible and there are several possibilities, it chooses the "most specific" possibility.) So a taxable item will use the `taxable-merchandise` method for **`sell`** and the `merchandise` method for buy.

Let us briefly return to the first example in this section, where we defined clos-stuff to behave differently on lists and non-lists. We provided the following definitions:

```
(defgeneric clos-stuff (x))
(defmethod clos-stuff ((x list)) (car x))
(defmethod clos-stuff (x) x)
```

And then we said: 'The first form announces a new generic function, called **clos-stuff**, which has a lambda list `(x)`. The next two forms describe the behavior of this function in two cases, if the argument *x* is a list, and if it is not.' It is easy to see how the first **defmethod** defines what **clos-stuff** does to a list argument since `list` appears right in the definition, but how does the second **defmethod** say what should happen on non-lists? It seems to be saying what to do for any argument, not just non-lists. The point is that CLOS checks all defined methods and picks the one that most closely matches the object of interest. If **clos-stuff** is passed a list, CLOS finds the method specialized to lists (the first **defmethod** form). If **clos-stuff** is passed a non-list, the method specialized to lists does not apply so CLOS picks the only other method (the second defmethod form).

13.5 Method combination

There is something unaesthetic about the **sell** method definition specialized to `taxable-merchandise`: it is almost word for word the same as the definition of the method for `merchandise`. This kind of repetition of code is inefficient and makes it hard to track errors and changes. Indeed, it is exactly what object oriented programming is meant to circumvent. CLOS has a way around these problems called *method combination*.

Let's look more carefully at the **sell** method for `taxable-merchandise`. Basically it is the same as that for `merchandise`, except that it does two more things afterwards: (1) it collects more money and (2) it keeps track of the increased tax liability. So it would be nice to be able to say (in effect): do everything as for merchandise and then do these two extra things. CLOS supports such a specification by allowing you to define methods that work in conjunction with other methods. These add-on methods can be applied before (a `:before` method), after (an `:after` method) or around (and `:around` method) the already defined methods. Since we want two things done after what is done to any merchandise, we define an `:after` method.

First, let's get rid of the last method we defined.

```
(remove-method (symbol-function 'sell)
   (find-method (symbol-function 'sell) nil
      (list (find-class 'taxable-merchandise))))
```

(We won't dwell on this form, suffice it to say that it removes the **sell** method specialized for `taxable-merchandise` that we just defined.)

Now, we will define an `:after` method for **sell** on `taxable-merchandise`:

```
(defmethod sell :after ((item taxable-merchandise))
   (incf *cash-on-hand* (* *sales-tax-rate* (price item)))
   (incf *sales-tax-owed* (* *sales-tax-rate* (price item))))
```

Now what happens when we apply **sell** to an instance of `taxable-merchandise`? CLOS searches through the methods that might apply to `taxable-merchandise`, either directly or by inheritance, and finds the two we defined. One, the `:after` method, is for `taxable-merchandise` and one is the inherited one for `merchandise`. Naively, one might think that only the former should apply, but it is an `:after` method, and can't be applied on its own, it has to follow something. So CLOS first applies the `merchandise` method and then the `taxable-merchandise :after` method, which is exactly what we want to happen.

```
(setq booze
   (make-instance 'taxable-merchandise
      :name "Wild Grandad"
      :price 25.0 :cost 12.0))
(buy booze 5)
(sell booze)
```

will increment our tax liability just as it should.

13.6 Other method types

As we said, there are `:before` methods as well as `:after` methods. The methods we initially defined, which weren't classified as `:before` or `:after`, are called *primary methods*. A `:before` or `:after` method can only be defined if there is already a primary

method defined which can be used in conjunction with it. (If you try to define a `:before` method without a primary method existing, a "method combination error" will be signalled.)

There are also `:around` methods, which are called both before and after other methods. How does this work? Suppose we, as proprietors of our store, decide to have a book sale, discounting our books by 20%. Suppose we had made this easy in advance by simply defining the class of books, which are taxable

```
(defclass book (taxable-merchandise) ())
```

and made sure that whenever we introduced a new book to our stock, it was entered as an instance of the `book` class.

```
(setq benji (make-instance 'book
              :name "The Tale of Benji"
              :price 35.00
              :cost 25.00))
```

To implement our discount, we could just write a new **`sell`** method for the class `book`, but this is unaesthetic and error-prone. Plus, we have to know exactly what is going on in the method, essentially meaning that we need access to the source code for **`sell`**. Instead, notice that all that is changing is the price of a book: let us define an `:around` method for **`price`**. Remember, **`price`** is not a blind function that just returns the price slot of a piece of merchandise: it is a generic function whose primary method is to return that slot. So we can combine it with `:before`, `:after` or `:around` methods, redefine a new primary method, whatever we like. In this case

```
(defmethod price :around ((a-book book))
   (* (call-next-method) 0.8))

(price benji) → 28.0
```

How does this work? The idea of an `:around` method is that it will be invoked before any of the other defined methods. Then, at some point in its execution, it can pass control to one of the other methods. It does this by invoking **`call-next-method`**, which invokes the "next" applicable method. The next method could be another `:around` method (for a more specific class) a `:before` method, or, as in our case, the primary method for the particular generic function. So `(call-next-method)` will return the slot-value of price for `benji` (35.00), and then this value will be multiplied by 0.8 and

returned. In this fashion, every book in the store will have its price reduced by 20%. Of course, when the sale is over, this method can be removed and books return to full price.

13.7 Shared slots and initialize-instance

There is a problem with using our work so far to track inventory: it doesn't really keep a list of what is in stock. One possible solution to this is to make a special variable (say `*inventory*`) to hold a list of all instances of merchandise. Another way, however, is to use *shared slots*. A shared slot is one whose value is common over all instances of a class. In this case, we would define a slot called `inventory` such that `(slot-value item 'inventory)` returned a list of all items, no matter what item was passed as an argument.

```
(defclass merchandise ()
   ((inventory :accessor inventory :initform nil
     :allocation :class)
    (name :accessor name :initarg :name)
    (price :initarg :price :accessor price)
    (cost :initarg :cost :accessor cost)
    (quantity :initarg quantity :initform 0 :accessor quantity)))
```

The only unfamiliar piece of this definition is the `:allocation :class` in the description of the `inventory` slot. This declares that inventory will be a shared slot.

What we want now is that every new instance of `merchandise` should be added to the inventory list. To do this, we use the built in generic function **`initialize-instance`**, whose mission is to initialize a new instance of a class. In particular, we can add an `:after` method to **`initialize-instance`** which simply pushes the new instance onto the beginning of the inventory list:

```
(defmethod initialize-instance :after
   ((item merchandise) &rest initargs)
   (push item (inventory item)))
```

Now, if we construct some new instances of merchandise:

```
(setq corn-flakes
   (make-instance 'merchandise :name "corn-flakes"
      :cost 1.00 :price 2.49))
```

```
(setq shaving-cream
   (make-instance 'taxable-merchandise :name "shaving cream"
      :cost 2.59 :price 3.49))
```

Each of these will have been added to the inventory list:

```
(inventory corn-flakes)
→ (#<TAXABLE-MERCHANDISE #x3763C8> #<MERCHANDISE #x376BDC>)
```

(Of course the old items we constructed aren't on the list because they were initialized before the inventory list was a slot.)

13.8 Beyond scratching the surface...

This introduction was meant only to give the flavor of CLOS. We have barely begun to describe generic functions, classes and methods, and have not even mentioned metaobjects, defining your own method combination types, or any of a host of other useful and interesting topics. These topics are explored in other, more exhaustive, treatments. And as always, the best way to understand how CLOS is used is to look at the source code for a well-written CLOS program.

Chapter 14 Errors

Any errors which occur during evaluation are signalled by a dialog box giving details of the error and requesting that some action be taken. Usually you are given the choice of aborting the evaluation or entering the Debugger, but it is also possible to continue in the case of some less serious errors. At this stage, you may find it confusing to enter the Debugger but we recommend that you try it; you can do no harm. A selection of typical error messages follows. These are accompanied by the code used to generate them, and a brief explanation of the problem. When the dialog box is cleared from the screen, the text of the message is printed into the Toploop window. The Debugger is described in chapter 6 of *Programming Tools*.

14.1 Typical error messages

Restarts dialogs

When an error (or any break in a program) occurs, a Restarts dialog is displayed. (In previous versions of Allegro CL for Windows, an simple error dialog was displayed). A Restarts dialog (several are illustrated on the following pages) displays the error message, identifies the condition signaled (unless it is `simple-error`), lists available restarts and has several button. The button include **Invoke Selected Restart**, **Enter Debugger**, and **Abort**.

Abort means clear the error and the stack and return to the Top Level. Any ongoing computation is canceled.

Enter Debugger means bring up a Backtrace window showing the state of the stack.

Invoke Selected Restart means take the action called for by the selected restart. Among the restarts usually available are Enter Debugger and Return to Top Level, which correspond to the two buttons just described. Continuable errors usually have a restart to continue computation, perhaps after providing additional information.

Lisp provides too many error messages for us to be able to reproduce every one here. We have therefore provided a selection of the ones you are most likely to encounter as you begin to try things out.

1. You can reproduce the error message below by typing:

```
a
```

In this instance, the variable a had not been assigned a value prior to this request to view its value.

2. The following message was caused by typing:

```
(jim 1 2)
```

Remember, Lisp always regards the first item inside the parentheses as a function call unless told otherwise.

3. The function **cons** expects two arguments: a symbol and a list. This error can be reproduced by typing:

```
(cons 'a 'b '(fred))
```

4. Consider this version of **my-log2**:

```
(defun my-log2 (num &optional base)
   (if (not base)
      (= base 10))
   (/ (log num) (log base)))
```

Trying to make the base equal to 10 with `(= base 10)` instead of `(setf base 10)` caused this problem. If you define **my-log2** as above and try to call it with argument 2, the following message appears:

5. Here is another version of **my-log2**:

```
(defun my-log2 num &optional base
   (if (not base)
      (setf base 10))
   (/ (log num) (log base)))
```

This error appeared because there are no parentheses round the argument list.

6. The message below resulted from this version of **sum-of-squares2**:

```
(defun sum-of-squares2 (&rest nums)
   (let (result 0)
      (for i in nums
         do
         (setf result (+ (* i i) result)))
      result))
```

Notice that the argument to **let** should be enclosed by two sets of parentheses.

14.2 User-defined error messages

You may also signal errors from your programs using the **error** function. The function should be called with something like:

```
(error "division by zero")
```

A more elegant version of the function **my-rem1** would be:

```
(defun my-rem2 (numerator denominator)
   (cond ((< numerator 1) 0)
         ((< denominator 1)
          (error "division by zero"))
         ((< numerator denominator) numerator)
         (t (my-rem2 (- numerator denominator)
                     denominator))))
→ MY-REM2
```

If you try out **my-rem2** with the arguments 2 and 0, the following error message will appear:

14.3 'Attempt to set non-special free variable' warning

This is a warning that you may see from time to time. Suppose the file *foo.cl* contains the two forms:

```
(in-package :user)
(setq foo 10)
```

And assume this is the first time the symbol `foo` has been used. When you compile or load the file, Lisp will print the following warning:

```
;;;; Warning: Compilation warning: Attempt to
set non-special free variable FOO. So will
treat as special.
```

So what is going on? A free variable is one that is not bound (e.g. in a **`let`** form) or an argument in a function call. Free variables should first be defined with **`defvar`** or **`defparameter`**, as follows:

```
(defvar foo 10)
```

or

```
(defparameter foo 10)
```

Either of these forms defines `foo` as a special variable. Now, forms like

```
(setq foo 20)
```

will not cause the warning.

These warnings are quite useful when compiling files, since they may indicate a coding error or a typo. They are annoying at the top-level (where you may enter **`setq`** forms on the fly to test or debug code). For that reason, the warnings have been turned off when typing to the top level, but maintained when loading or compiling files. (All files are compiled before loading, so the only difference between loading and compiling is whether the file is loaded.)

If you see these warnings and wish to get rid of them, put the appropriate **defvar** or **defparameter** forms in the file. (You can turn off all compiler warnings with the dialog displayed by choosing Compiler and CLOS in the Preferences menu.)

Differences between defvar and defparameter

There is an important and useful difference between **defvar** and **defparameter**. Each takes a value argument, and the first **defvar** or **defparameter** form sets the variable to that value. However, subsequent **defvar** forms do not change the value while subsequent **defparameter** forms do. Thus:

```
(defvar foo 10)
foo → 10
defvar foo 20)
foo → 10

(defparameter bar 10)
bar → 10
(defparameter bar 20)
bar → 20
```

This can be a very useful distinction. You may have several files where variables are defined. Using **defvar** and **defparameter** as appropriate, you can arrange that the first definition (in the first file loaded) or the last definition (in the last file loaded) actually sets the value after all files are loaded.

[This page intentionally left blank.]

Appendix A Glossary

Glossary of terms used

Access Function

■ A function which directly accesses data structures to build, modify and find items. Access functions are called by other functions to manipulate the data. If the structure of the data is modified, only the access functions should need to be changed.

Array

■ A dimensioned data structure of cells into which information is stored.

Assignment

■ The process of allocating a value to a symbol or list.

Association List

■ A list which contains a number of sublists, each of which has a key as the first element.

Binding

■ The process of assigning a particular value to a variable within a specific section of code.

Call

■ A request to Lisp that a function be invoked.

Clause

■ A list of forms, one to test and the other to be done if the test is successful, following **cond**.

Data Abstraction

■ The concept of producing software which is independent of the data format. The object is to create as few functions as possible to access the data directly and to make all other functions call these. In this way, only these access functions need to be changed if the structure of the data changes.

Dotted Pair

■ Special notation to indicate that the second argument to a CONS was not a list.

Dynamic Scoping

■ Means that the value accessed by a variable is that most recently bound during the execution of the program, c.f. lexical scoping.

Element

■ An item in a list. This may be a symbol, a number, or another list.

Evaluation

■ The process by which Lisp assigns a value to an expression.

Filtering

■ The process of examining lists and deciding whether or not to discard elements.

Form

■ An expression which is evaluated to return a value.

Free

■ The term used to describe a variable which is used within a function without establishing a new binding.

Function

■ A section of code which expects arguments to be passed to it. These are then manipulated to return a value.

Garbage Collection

■ A process, normally initiated automatically by Lisp, to retrieve free space which had been used as work space by certain functions.

Iteration

- The repeated execution of a particular section of code.

Key

- The first element of a sublist in an association list sublist, which is used to find that sublist.

Lambda Expression

- A 'temporary' function which is usually called from inside another function. It does not have a name and is 'discarded' once it has been evaluated to return a value.

Lexical Scoping

- Means that the value accessed by a variable is that governed by the nearest textually enclosing binding in the source code of the program, c.f. dynamic scoping.

List

- A collection of elements enclosed in parentheses.

Literal Expression

- An expression that is not to be evaluated.

Macro

- A method of writing sections of code which are used repeatedly in a shorthand form.

Macroexpansion

- The process of converting a macro call into the actual form for evaluation.

Mapping

- The application of a function separately to each of the elements in a list.

Multiple Escape Characters

- Vertical bars which are placed round a symbol to allow certain special characters to be included in it.

Multiple Values

■ Some Lisp functions may return more than one value as their result. In most situations, values other than the first are discarded, but special functions are available to access those other values.

Numbers

■ Numerical values which fall into four categories: integers, floating-point numbers, ratios and complex numbers.

Predicates

■ Functions which perform tests to return one of two values: `t` (true) and `nil` (false). (A predicate may be defined to return a non-`nil` value other than `t` when that value may be useful.)

Property

■ An attribute consisting of a name and a value which can be assigned to a symbol.

Property List

■ A collection of items which are properties.

Property Name

■ The symbolic name of an attribute which is assigned as a property of an element.

Property Value

■ The value of an attribute assigned as a property of an element.

Recursion

■ A method of programming during which a function calls itself repeatedly to assist in the solving of a problem

Scope

■ The range of code during which a particular value is accessed by a variable.

Side Effect

■ Something, other than the returning of a value, which may occur as a result of evaluating an expression.

Single Escape Character

- A backslash (\) which is used to suppress a special character when printing.

Special Variable

- A Lisp variable which uses dynamic scoping.

Symbol

- One of the fundamental Lisp data types, an indivisible word-like object.

Textual Scope

- The same as lexical scope.

Top-level Element

- An object which is at the highest level of a list, i.e. it is not a list within that list.

Toploop

- The process by which user interaction is handled by Lisp. It accepts input, evaluates it and prints the returned value.

Terms used in other texts

Atom

- Any object other than a list or `nil`. 12 and `'a` are both atoms.

Data Constructor

- An access function to build a data structure.

Data Mutator

- An access function to modify the data within a structure.

Data Selector

- An access function to find items within a data structure.

Embedded Function

- A call to a function which appears inside another function.

Environment

- A collection of bindings.

Functional Error

- A bug which causes the code to do something different from what we want, without necessarily producing an error message.

List Surgery

- Applying functions to lists to modify their contents, such as **`append`**, **`first`**, **`subst,`** etc.

Nested Expression

- An expression within an expression, such as a list which is one element of a list.

Pretty-printing

- A process by which line breaks and indentation are inserted into code to make it more readable.

Primitive

- A built-in Lisp function.

Procedure

- In Winston & Horn, a function written by the user.

Sentinel Value

- A value which is input to terminate a loop.

setq

- A low-level function for updating variables. Other texts use it in place of **`setf`**.

  `(setf x y)` ≡ `(setq x y)`

  but **`setq`** is less powerful than **`setf`** in that it can only update variables.

Tail Recursive

- A recursive function which produces a solution which is fully formed when it is found and therefore does not need to be passed back through each level of recursion for further computation.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Allegro CL for Windows

Interface Builder

version 3.0

October, 1995

Copyright and other notices:

This is revision 1 of this manual. This manual has Franz Inc. document number D-U-00-PC0-12-51018-3-1.

Copyright © 1994 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademarks of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, Windows 95, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

Contents

Chapter 1 Getting started with the Interface Builder

1.1 Introduction

The *Interface Builder* (or *IB* or *Builder*) allows you interactively to construct the graphical user interface to your application. You use the mouse to create, position, and resize widgets (dialog-items) as well as dialogs and other windows. Attributes of widgets and windows are edited by using either pop-up menus or special dialogs that list all their modifiable attributes. Once you have created a complete dialog with widgets or arbitrary window hierarchy, the Interface Builder generates the source code needed to recreate it programmatically and saves it to a standard Lisp source code file that you include in your application.

A variety of features allows quick interface creation. For example:

- Each of the standard Windows widgets can be created from a floating palette or from a pop-up menu, and then moved or resized at any time.
- The event handler Lisp code associated with a particular widget, such as the function that runs when the widget's value changes, can be located quickly from the widget itself.
- Widgets can be easily aligned with each other. While moving a widget, it “snaps” into alignment whenever its edges are nearly aligned with the same edges of other widgets. Red alignment lines are drawn to show which widgets are lined up.
- Widgets can be cloned and/or dragged to other dialog windows.
- Individual attributes can be copied from one widget to another.

- Groups of widgets can be moved, cloned, or deleted at once.
- The auto-generated source code can be edited by hand later if you wish, and the dialog created from this edited code can still be further edited using the Interface Builder.
- You can instantly switch any window between edit mode and run mode, allowing frequent testing of your changes.

1.2 Organization of this manual

Documenting a mouse-based tool is always somewhat complicated, because users have many choices and their actions rarely follow a set pattern (first do this, then that, then this other etc.) What most people want to know is how to get started and then how to do specific things. We have tried to organize the manual in that way. Getting started and some basic concepts are described in this chapter. Chapter 2 has a long example with heading describing what is being done in each step of the example. We hope those headings will include things you want to do. Chapter 3 describes specific mouse actions and specific menus.

1.3 Accessing the Interface Builder

The menu commands for the IB are grouped on the Builder menu.

The first three items allow you to create windows and switch between editing them and testing them.

The rest of the items simply expose each of the Interface Builder's own dialog windows, which you use to create your own window interface.

1.4 Some general concepts

- **Run mode and edit mode**. A dialog which is being constructed can be in run mode or in edit mode. When it is in run mode, it is just a normal dialog, meaning that mouse and keyboard actions have their usual programmatic effect. When it is in edit mode, dialog items (widgets) are not sensitive to the mouse or the keyboard in the usual way. Instead, the mouse and the keyboard are used to modify the widget in some way. It is typical to switch between run and edit modes while building a dialog, on order to test the features you have added to the dialog.
- **The set-value-fn is important**. A dialog must do something programmatic (unless it is simply informative or decorative). When a user uses the mouse or types to the keyboard over a dialog, what typically happens is that the `set-value-fn` associated with the dialog item being operated on is run. It is that function that does the work of effecting the action of the dialog. The default set-value-fn is **`true`**, which does nothing so you typically will want to define your own.
- **Widget and dialog-item mean the same thing**. As we use the terms, they are synonyms.
- **The status bar is very important**. The status bar is used extensively by the IB. Information about the objects you are creating, about the contents of the IB palette, about what the IB is doing, and about what you can or should do next are all displayed in the status bar. Therefore, it is essential that the status bar be present and visible. If the status bar is hidden, choose **Status Bar** from the Tools menu to cause it to appear or press the F11 key. (The F11 key toggles the status bar between visible and hidden. Because the status bar is always above most windows and dialogs, the F11 key is quite useful when using the IB.)

1.5 Things to note that you might not expect

- **Alt-Tab does not wor**k when the mouse cursor is over a window being edited. Normally you can press Alt-Tab to switch from Allegro CL to other Windows applications. However, when the mouse is over a selected window that is in edit mode (so that the mouse cursor is a cross), Alt-Tab has no effect. To use Alt-Tab, first move the cursor out of the window being edited.

- **Only the left mouse button can be used** to choose menu items. Even though the right mouse button is used to pop-up menus in the Interface Builder, the right button cannot be used to select items on the menus. You must first release the right mouse button and then left-click on a menu item in order to choose it. To get rid of the menu without selecting an item, left-click somewhere off the menu.
- **Menu bars appear only on top-level windows**. If you use the menu editor to add a menubar to a window, you won't see the menu bar until you make the window into a top-level one. A top-level window is a window whose location (the value of the *location* argument to **open-stream**) is *screen*.
- **Some attributes require that a widget or window be recreated** in order to change that attribute. You may not notice when the Builder does this, but if you have a global variable bound to a widget or window and then change one of these attributes, your global variable will then be bound to a closed stream or other bogus value.
- **You should save code to generate the new interface to a file**. Information on saving code is given under the heading How to save the auto-generated code for recreating edited windows in section 2.1 **Tutorial**. In brief, click right over the background of a dialog window, choose **Code** from that menu and **Save Code To File** from the submenu. A shortcut is to left click in the background of the window while the Control key is down.

1.6 Interface Builder Dialogs

The Interface Builder has 5 special dialog windows of its own to aid you in creating yours. These are the *Widget Palette*, the *Widget Editor*, the *Window Editor*, the *Menu Editor*, and the *Builder Preferences* dialog.

Each of the Builder dialogs except the Preferences dialog can be selected from the Builder menu, and the three editor dialogs can alternately be selected from the IB's pop-up menus. The Preferences dialog is displayed by choosing **Interface Builder Preferences** from the Preferences menu.

The IB dialogs will take a bit longer than usual to come up the first time you select them, since they are each created “lazily” the first time they are needed, as are most of the system windows in Allegro CL for Windows.

Each of the Builder dialogs can be hidden from view by closing them from the system menu box at the upper left corner of the window. Either double-click the box, or single-click it and select **Close** from its menu. Or just press control-F4 when the window is selected. The IB dialogs are not destroyed when you close them, as windows are by default. They are simply made invisible, and will come up more quickly when you call on them the next time.

1.6.1 The Builder Preferences Dialog

The Builder Preferences dialog allows you to specify various parameters that control the operational style of the Builder. The only way to select this dialog is by choosing **Interface Builder Preferences** in the Preferences menu, or its keyboard shortcut (Control-Shift-F). When you alter the value of widgets on this dialog, the changes take effect immediately; there is no need to click on an Apply button as with the editor dialogs or to close the preferences window (though you'll likely want to close it to get it out of the way). For the few text-widget items, you will need to tab out of the item or select another window before the change takes effect.

Note that the IB preferences are handled separately from the preferences for the rest of Allegro CL for Windows, which is why they are not part of the standard Preferences dialog. They are saved to a separate file than the other Allegro CL preferences. To cause the Builder preferences to be loaded automatically when Allegro CL is launched, you should save them into a file called *ibprefs.lsp* in the directory that contains the *lisp.exe* that you run. (The other Allegro CL preferences are loaded from *prefs.lsp* in this same directory.) Use the **Save** at the upper right corner of the Builder Preferences dialog to perform the save.

Here is the preferences dialog, followed by brief descriptions of the various widgets on it. (Note that we have produced a monochrome picture for readability but much of the 3-D detail is lost.).

Drag Actual Widgets (rather than an outline) Default: On

When you move and stretch widgets interactively, the actual widget will be dragged if this option is on. Otherwise you will drag a simple black rectangle, and the widget will jump to its new location once you click the rectangle into place. While moving the widget itself is more intuitive and slicker-looking, moving the black rectangle may be smoother on some machines, allows you to see where the widget was during the move, and lets you align the widget with where it was (in case you want to move it either vertically or horizontally only).

Sticky Move Horizontal /Vertical Default: On

These two check boxes specify whether the sticky alignment feature is in effect in the horizontal and vertical directions independently. You may want to disable stickiness if you want to position widgets where they happen to be nearly but not exactly aligned with other ones. Alternately, you can toggle horizontal stickiness off and on by pressing the control key while dragging a widget, and likewise toggle vertical stickiness by pressing the shift key while dragging.

Sticky Distance Default: 4

Specifies the number of pixels away from alignment within which the moving widget must be in order for it to be snapped into exact alignment, assuming that sticky alignment is currently turned on.

Min Pixels Btwn Widgets Default: 0

By default (with this option set to zero), when you move or resize a widget so that it overlaps other widgets or subwindows, it is then shifted automatically so that it butts up against the other objects but without overlapping them. (Sometimes the Builder doesn't find an obvious non-overlapping spot at which to place the moving widget, in which case it will remain overlapped and a beep will sound.)

If you set this option to a positive number, then the moving widget will be moved even farther so as to maintain at least that many pixels between it and other widgets. If you want to place a series of buttons exactly 8 pixels apart, you could set this option to 8 and then move the buttons near each other one at a time and let the Builder set the distance between them to be 8 pixels.

If you set this option to a negative number, then widgets will be allowed to overlap by a number of pixels equal to the absolute value of the number. You can also set this option to `nil`, in which case all overlapping constraints are removed completely.

Confirmation: Deleting a Widget Default: On

When you delete a widget or group of widgets interactively with the Builder, you will first be prompted for confirmation with a pop-up dialog if this option is selected. Otherwise the widget(s) will be deleted without hesitation. Accidentally deleted widgets can still be recreated via the Undo item on the Window Pop-up Menu.

Confirmation: Overwriting Unaccepted Edits Default: On

Sometimes a widget is read onto the Widget Editor even though the modifications for the widget that is currently being edited there have not yet been applied. If this option is on, you will first be asked if you want to apply those changes to the current widget before editing the new one, or discard the changes, or cancel reading the new widget onto the Widget Editor. If this option is off, then the modifications for the current widget are discarded.

Confirmation: Exiting with Modified Windows Default: On

If you try to exit Allegro CL while there are windows that you have edited with the Builder and that haven't been saved to file since your most recent changes, then you will first be prompted for confirmation in a pop-up dialog if this option is on. You can either have the dialogs saved and continue exiting Allegro CL, discard the modifications since the last save, or cancel exiting Allegro CL. If this option is off, the modifications will be discarded and Allegro CL will continue to exit.

Break on User Errors Default: Off

If you make certain errors while using the Builder, such as specifying an invalid value for a widget attribute, this option determines how you will be notified. If it is off, a beep will sound and a message will be printed in the Allegro CL status bar, and the current operation will continue normally. If it is on, then a Lisp break will occur with the usual Restarts dialog including options to abort or debug, and the current operation will be interrupted.

Use Cascading Submenus Default: On

When you pop up the right-button Window Menu or Widget Menu, the submenus of that menu will cascade to the side if this option is on. If it is off, the submenus will pop up independently in the same location. Cascading menus allow you to browse the whole menu hierarchy, but force you to move the mouse further in order to select an item on a submenu.

Object Under Mouse Messages Default: On

When the mouse cursor moves over widgets in a dialog window that is in edit mode, if this option is on then help messages will be displayed in the Allegro CL status bar indicating what each type of mouse click will do if performed on the object under the mouse. After you become familiar with these actions, you may want to turn this option off, because these messages cover over other informative messages that the builder displays after completing various operations. A single object-nonspecific message will still be printed when the mouse enters a window that is in edit mode.

Allow Editing of System Dialogs Default: Off

When you select a window to edit with the Builder, you normally can choose only from the set of windows that were originally created with the builder. If this option is selected, you will also be able to select any other dialog window, includ-

ing Allegro CL system dialogs. The additional windows will not appear on the pop-up menu of windows to edit, but you can select them by mousing them directly instead of an item on the menu.

While modifying the Allegro CL system dialogs is not supported, you can use this option to explore them at your own risk.

Use Arrow Cursors Over Widgets Default: On

If this option is on, the mouse cursor will change while moving it over a widget on a dialog that is in edit mode to tell you which move or stretch operation is possible at each point over the widget. If you find the frequent cursor changes annoying, you can turn this option off, and the cursor will remain a cross when moving over widgets.

Use Neater But Slower Code Printer Default: On

When the Builder generates the Lisp source code to recreate your windows, it pretty-prints the code to a file. Standard pretty-printing puts the code into an easily-readable format generally, but doesn't arrange each attribute name and value onto its own line of text. If this option is on, these attribute plists will be further arranged this way for easy editing. This slows down the code saving quite a bit, so you may want to turn this option off to speed up saving your windows to file. Or you could turn it off during development until if and when you decide that you want to view or edit the code file, and then turn it on for a final save before you view the code.

Read New Objects Onto Editor Forms Default: On

If you create a new widget or window and the editor dialog for that type of object is not hidden at the time, then the new object will automatically be read onto that editor dialog if this option is on.

Read Moused Objects Onto Editor Forms Default: Off

If this option is on, then anytime you click on a widget or window that is in edit mode and the editor dialog for that type of object is currently not hidden, then that object will be automatically read onto the editor dialog.

Save Current Widget Values with Dialog Code Default: On

By default, when you save an edited dialog to file, all of the current attributes of its widgets are recorded in the file, including their current values. If this option is off, then the values for text widgets are saved as `nil` or "" as appropriate rather than saving their currently displayed strings.

Code File Columns Default: 72

When the Builder generates the code for recreating a window and saves it to file, the Lisp forms will be pretty-printed with a maximum (where possible) of this many characters per line.

The font buttons on the Preferences dialog

The three Default Font buttons (for fixed, proportional, and bold fonts), when clicked on, display a choose-font dialog that allows you to choose the default font of the appropriate type (but no check is made to ensure you have specified a fixed, proportional, or bold font). The current fonts are displayed in the button labels.

Chapter 2 Tutorial

This chapter contains a long tutorial, with examples of how to do many things. Then at the end of the chapter are several sections with additional useful information.

2.1 Tutorial

In this tutorial, we create a dialog and do various things with it. You, of course, will likely want to create a different dialog, but you may want to do things similar to the things done here. Even if you do not want to follow the tutorial, you may find a heading that describes something you want to do. Reading the information under that heading (and perhaps some preceding headings) should tell you how to do what you want.

How to create a dialog window

Choose **Create Window** from the Builder menu. Another menu will pop up displaying the classes of windows that can be created with the IB. The most interesting of these is **Dialog**, since that's the only type of window on which you can place widgets. (If you programmatically create subclasses of any of the displayed window classes, they will thereafter appear on this pop-up menu of window classes that can be created and edited with the IB.)

Choose **Dialog** from the submenu.

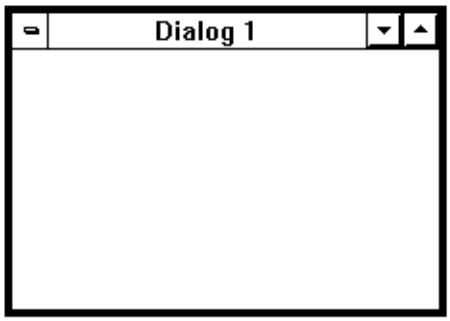
Place the new dialog by pressing the left button down where you want the upper-left corner to be, and then dragging the mouse to where you want the lower right corner to be, releasing the button at that point.

Note that the interface builder prints many prompts in the Allegro CL status-bar, including some prompts that are visible only while the mouse is held down, such as when you drag out the lower right corner. When you release the mouse, a new dialog window will appear on the screen.

An alternate way to create a window is to click the Create Window button (it has a picture of a blank window) on the Allegro CL toolbar.

Note that when you move the mouse cursor over your new window that the cursor changes to a thin black cross. This indicates that the window is currently in *edit mode*. While in edit mode, mouse clicks do not perform their usual behavior as defined for your application, but rather perform special functionality as defined by the interface builder itself.

A blank dialog looks like the following:



The dialog menu: what it is and how to display it

Click the right mouse button over the interior (client) area of your new dialog window. The pop-up menu for editing windows will appear. You can release the mouse button to peruse the menu since you need to click again to select an item. Left-click on **Set Attribute**, **Main Attributes**, and then **Title**. When the small dialog pops up, type a new string to display in the title bar of your new window. When you press return, the new title appears on the window. We give our dialog the title “New Dialog”.

How to create a widget

Go back up to the Builder on the Lisp menu bar, and select **Palette** from the menu. After a moment, a palette showing each of the standard types of widgets will appear. (We use the term “widget” interchangeably with the term “dialog item”.)

Button	Default-Button
Cancel-Button	Picture-Button
Radio-Button	Check-Box
Editable-Text	Lisp-Text
Multiline-Editable-Text	Multiline-Lisp-Text
Static-Text	Combo-Box
Single-Item-list	Multi-Item-list
Outline	Dropping-Outline
Lisp-Group-Box	Tab-Control
Horizontal-Scroll-Bar	Vertical-Scroll-Bar
Progress-Indicator	Track-Bar
Up-Down-Control	Static-Picture
Header-Control	Grid-Widget (**Professional only**)

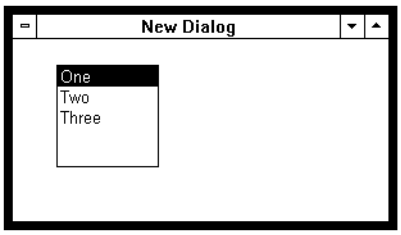
On the widget palette, locate the single-item-list button near the middle left. It’s the one with the label in **bold**. Of course, these labels do not appear on the screen, but as you move the mouse over the palette, the type of widget is printed in the status bar.

Click on the palette's single-item-list button now. The item will stay depressed as it waits for you to position a new list widget, to remind you what you are creating.

Click the left mouse button down (and don't release yet) in the interior of the dialog window that you created earlier. After a brief moment, a list widget will appear with its upper left corner where you clicked. While still holding the mouse button down, you can option-

ally drag the lower-right corner of the widget in order size it with the same click with which you positioned it on the window.

Here is our dialog with the new widget.



How to move a widget

Move the mouse over the new list widget. Notice that the mouse cursor changes depending on what part of the widget it is over. This indicates what a left-click will do at that particular spot on the widget. The four-directions arrow in the middle of the widget indicates that you can move the widget by clicking and dragging near its center. Try this now. The other arrows around the edge of the widget indicate which one or two directions you can stretch the widget in at each spot.

When you move the mouse over the widget (while its dialog is in edit mode), a message is displayed in the Allegro CL status bar indicating what other mouse buttons, sometimes combined with shift keys, will do. These other clicks will have the same effect regardless of what directional cursor is showing over the widget; the simple left-click is the only one that depends on the position of the cursor over the widget.

Switching between edit mode and run mode

Choose **Run Window** from the Builder menu. This will pop up a menu containing the windows created by the IB (in our example, the menu will contain only our new dialog, since that is all we have created so far). You select a window by either choosing it on the pop-up menu, or by clicking directly on it.

Do this to the new dialog now.

Now when you move the mouse cursor over the new dialog, the usual arrow cursor appears indicating that the window is in *run mode*. This is the normal mode for windows and in this mode, the mouse has the usual effect (rather than being an editor tool).

Click an item in the list widget on the new dialog. As you expect in run mode, the item clicked over is selected. In fact, you are already running your application at this point, and if the list widget had a set-value-fn attached it would be invoked already. One of the strong points of the IB is the ease with which you can switch between modifying and testing your application. Dialogs or other windows can be individually switched between edit and run mode.

Other ways to switch between edit mode and run mode

- **Use the Run Window and Edit Window buttons on the Allegro CL toolbar** (they are next the Create Window button you used above -- the status bar describes the buttons on the toolbar as you pass the mouse over them).
- **Use the keyboard shortcuts for the items on the Builder menu**. Typing Ctrl-Shift-R while the mouse is over a window puts the window into run mode at any time. Ctrl-Shift-E puts it into edit mode. If the mouse cursor is over an editable window when you type Ctrl-Shift-E, it will begin editing that window immediately. Similarly with Ctrl-Shift-R. If the cursor is not over an editable window, the key combinations pop up a menu listing windows (the same behavior as choosing the menu items, as described above). If the window under the mouse was not selected, it is selected automatically at this time. Click over -- i.e. select -- the window if it does not go into edit mode. Edit mode is in effect only when the window is selected.

The mouse cursor tells you the mode

When a window is in edit mode, the cursor is a thin black cross or a double or four-way arrow. When a window is in run mode, the cursor is the normal cursor (usually an northwest arrow).

Other ways to create a widget

An alternate way of creating a widget is from the window menu (in case you don't want to bring up the widget palette just to add a couple of widgets). Right-click on the background of your dialog window, select **Create Widget** from the menu that pops-up. Submenus will display the widget choices.

Choose **Buttons**, and then **Picture Button**. Then click and drag on the dialog window background to position the new picture-button widget just as when using the widget palette.

The ‘sticky’ widget alignment feature

Left click near the center of the new picture-button widget and drag it around the list widget. Notice what happens whenever an edge of the picture-button is nearly aligned with an edge of the list: the moving widget is automatically scooted over just a bit so as to be exactly in line with the other widget, and one or more red lines are drawn to indicate the particular alignment in effect. This type of alignment gives you better control than aligning to an arbitrary grid. Sticky alignment also works when resizing widgets.

While stickiness is usually handy, you sometimes want to position a widget so that it happens to be nearly aligned with another without it annoyingly snapping into alignment. For these occasions, there are a couple of ways to disable the stickiness.

- If you want to turn it off for a while or permanently, you can do so on the IB's Preferences dialog. (Choose **Preferences** from the Builder menu. The Sticky Move Horizontal and Vertical choices control sticky alignment in the two directions. The same dialog allows you to choose the number of pixels away that triggers stickiness.)
- You can temporarily disable stickiness with the Control and Shift keys. After you have left-clicked to begin moving or resizing a widget, press the control key at any time to toggle horizontal stickiness, or press the shift key to toggle vertical stickiness.

Using the widget menu

If you right-click over a widget in a dialog in edit mode, the widget pop-up menu appears. Following our example, right-click on the single-item-list widget that you created and select **Set Attribute**, then **Class-Dependent**, then **Range** (the contents of the menus are widget-specific). A pop-up dialog is displayed, showing the default range that the IB uses for list widgets, namely the list `(:one :two :three)`. Position the text cursor just inside the right parenthesis and add another element, such as `:four`. Then either click the OK button or press Alt-O to accept the new range value. The element that you added to the range will appear in the widget. If it doesn't fit, resize the widget as necessary.

Using the editor dialogs

Up to now we have changed attributes of a window and a widget by using their pop-up menus. An alternate technique for editing these objects is by using the IB’s special dialog windows that display all the attributes and their values at once. These dialogs give you a

better overall view and allow changing several attributes quickly, while the pop-up menus may be handier for making one or two quick changes without bringing up the larger editor dialog.

Right-click on the single-item-list widget and select **Edit on Form** from the menu that pops-up. After a moment, the Widget Editor dialog appear, displaying information about the widget that you clicked. (The keyboard equivalent is to left click on the widget while holding the Alt key down.

Here is the Widget Editor (as a monochrome picture, so 3-D details are lost):

The large list at the bottom of the Widget Editor displays all of the attributes that can be modified by the IB for the type of widget that it is currently displaying. Most of these attributes correspond to the initargs that you can pass to **`make-dialog-item`** when creating widgets programmatically.

To continue our example, click on the **Border** entry at the top of the list. The value for that attribute, namely `nil`, will appear in the single-line text widget just above the list of attributes. You can select any attribute from the list this way and edit its value by typing into the single-line text widget. Type `:none` now to remove the border. Notice that the change doesn't take effect yet, though; you still must click on the **Apply** button on the Widget Editor to apply the changes that you've made to the widget. Do this now. The border of the widget disappears. The Apply button is default-button widget, so you can also apply your changes by simply pressing the Enter key. After accepting your edits, the Apply key will gray to reassure you that you don't have un-applied edits remaining.

An alternative way to enter a new value for an attribute is to use the **Modify** button, just above the single-line text widget where you type new values. Click the **Modify** button now and a menu of choices appears. If the value were a Boolean (so `t` and `nil` are the only possible values), clicking on **Modify** would simply toggle the value.

The **Modify** button will perform various kinds of alternative editing depending on the type of attribute that is currently selected. Another attribute with multiple possible values is **Bottom-Attachment**. Click on it, and then click the **Modify** button again. A menu pops up with the possible values for bottom-attachment. (If you select `:bottom` from the pop-up menu, and then click the **Apply** button to accept that change, then the list-box widget will maintain a constant distance between its lower edge and the lower edge of its parent dialog window whenever that window is resized.) Other attributes that have long arbitrary values, such as **Range**, will pop up a dialog with a multi-line-editable-text widget when you use the **Modify** button. Some types of attributes have no extended editing, and the **Modify** button will be grayed out when they are selected.

The **Modify** button invokes what is loosely called extended editing of an attribute. Another way to invoke extended editing is to double-click the desired item in the large list of attributes on the Widget Editor. Double-click on the **Background-Color** item now and the Color Editor dialog will pop up, since background-color is a color attribute. If you change the color with the Color Editor and click its **OK** button, note that the color of your list widget doesn't change yet, because you still always need to click the **Apply** button on the Widget Editor (or press Enter) to apply the changes.

You may notice that the Widget Editor automatically updates itself if you change the widget that it is displaying in some other way, such as by moving the widget or changing an attribute with its pop-up menu. If you're not currently using the Widget Editor, you may want to close it (using the close box at its upper left) in order to avoid any redisplay delays as the Widget Editor updates itself.

How to apply the widget editor to a widget

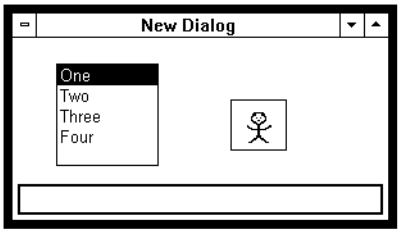
If you display the widget editor by choosing **Edit on Form** from the menu that pops-up when you right click over a widget, the editor will edit that widget. If the editor is already visible, you can click on **Select** and then click on a widget. The edit will then edit the selected widget.

How to set widget event handlers

So far we've just been changing the appearance of the application interface. This is all well and good, but not entirely useful unless the widgets can be made it do something. The IB can help in this area also, by making it easy to quickly locate the underlying lisp code for each widget that you write to respond to various events for that widget.

What we will do in our example is add a status bar to the dialog, and then arrange it so changing the value of the single-item-list widget prints a message to that status bar. Of course, your application will likely want to do something more complicated than printing a message, but that can be arranged by having the function you write do something other than print a message. The important thing is knowing how to set the function.

We need a bit of preparation for the following example. Select our example dialog and make sure it is in edit mode (the cursor will be a thin cross -- if it is not, put it in edit mode now by entering Ctrl-Shift-E or choosing **Edit Window** from the Builder menu). Right-click the dialog's background, and, from the menu that appears, select **Set Attribute**, then **Flags**, then **Status Bar**. A status-bar will be added to the dialog, useful for displaying messages to the user that pertain to that window. Here is our dialog with the status bar (and the picture button we added some time ago).



Now return to the Widget Editor dialog. It should still be displaying your single-item-list widget. (If not, click the **Select** button and then click on your single-item-list widget to read it back onto the Widget Editor.) The event handlers for a widget (such as set-value-fn, set-focus-fn, and mouse-in-fn) are treated just like other types of attributes, so scroll the Widget Editor's list of attributes down to the item **Set-Value-Fn** and click this item. Its default value is the function name **`true`**, which tells the widget to do nothing when its user value changes. Type the symbol `my-set-value` into the single-line text widget where it says `TRUE`, and press Enter. Your list widget will now try to run the function **`my-set-value`** when its value changes. Click the **Modify** button. Since the symbol `my-set-value` has no function binding yet, a new Lisp-editor window will appear with a skeleton definition

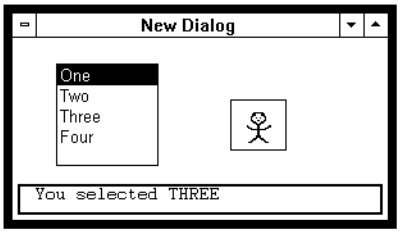
for this function, complete with the parameters that are needed for a set-value-fn function. (If the function already existed, a find-definition would find the existing version instead.)

The text cursor will be positioned where you can immediately begin entering your own code. Insert the following form at the cursor:

```
(window-message (dialog-item-dialog widget)
                "You selected ~a" new-value)
```

Now evaluate the definition (you can do this by typing Control-D if you are using host mode in your editor windows, which is the default). Then click on your dialog and put it into run mode (remember that you can do this by typing Control-Shift-R with the mouse over the dialog, or use the Allegro CL menubar or toolbar). Then click to select a new item in your single-item-list widget, and if these instructions are adequately written then a message will be printed in the status-bar of your dialog. So that's how easy it is to create a running application!

Here is the dialog with the message displayed:



An alternate way to find the source code associated with a widget is to use the right-button widget pop-up-menu, where the event handlers are listed under **Set Attribute** and then **Event Handlers**. To find the set-value-fn in particular, you can use the shortcut of holding the Control key down and left clicking the widget.

How to save user-written code for widget event handlers

Even though the IB created the skeleton code for the widget's event handler function and gave a default filename to the new Lisp-edit window that it created for it, you can save the code wherever you want, as with the rest of your source code. And while the IB will put all new widget event handler functions that it creates for a given dialog into a single window, you can move individual functions from there to various other buffers if you want. Just remember that you will need to evaluate the function in its new window afterwards (or load

its source file) in order for **find-definition** to find the code in its new place. If you want to avoid moving the definitions to your own files, just define the functions yourself in the appropriate files before using the IB to locate them.

How to save the auto-generated code for recreating edited windows

Once you have interactively created the windows and widgets that you need, they must be saved to disk in some way so that your application can recreate them later programmatically. The IB accomplishes this by writing Lisp source code that recreates each window to a standard *.lsp* file. The source code uses user-level Common Graphics functions such as **`open-stream`**, **`open-dialog`** and **`make-dialog-item`**, just as you might use yourself to create windows and widgets programmatically.

To save the source code for your dialog window, click right on its background and select **Code** from the window pop-up menu and then **Save Code to File**. Since you haven't yet established what file to save the source code in, the file selection dialog will pop up to prompt you for a pathname. After you enter the pathname once, it will be remembered for future saves, though you can specify a different path later if you want.

While the source code is being written, the cursor will change to an hourglass and the Allegro CL status-bar will indicate that a save is taking place. This operation can take a while, especially if you have a lot of picture-button or static-picture widgets on a dialog, since the bits that comprise the images for these widgets are saved as part of the window's code file so that you don't have to load them from various other files at runtime.

A shortcut for saving a window's source code is to hold the control key down and left-click the background of the window. You may wish to do this fairly often, just as you do with source code buffers.

When the save operation has completed, right-click the dialog again, select **Code** and then **View Code File**. A Lisp-edit window will pop up displaying the source code that the IB generated for this dialog window. While it is the goal of the interface builder to generate code that requires no further editing on your part, you can still always edit the generated source code directly if you want. After you have recreated the window by loading the edited code, you can once again edit the window interactively and save it with the Interface Builder.

If you create a hierarchy of windows with the IB, you need only save the uppermost parent window to disk, since code will be generated for all of its child windows along with the

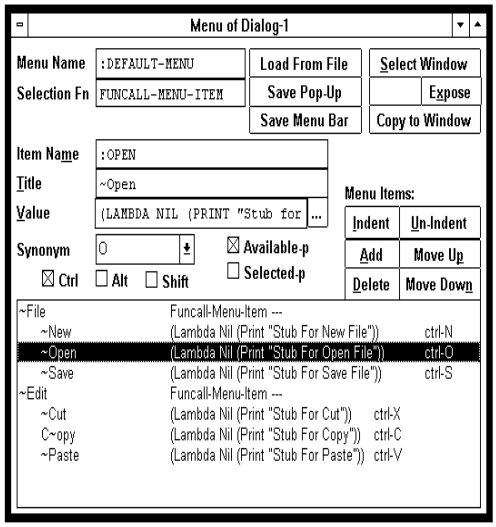
code for the parent itself. You could still save code for a sub-hierarchy independently if you want, though that shouldn't be necessary.

The code for the dialog we have created is at the end of this chapter.

How to create and edit menus

In addition to editing windows and widgets, the IB can also be used to create and edit menu bars and pop-up menus. Right-click the background of your dialog and select **Edit Menu Bar** from the pop-up window menu. The Menu Editor dialog will appear. Since your window doesn't have a menu bar yet, the Menu Editor will display its representation of a default menu, containing two items called **File** and **Edit**, these each containing three sub-items.

Here is part of the menu editor:



As you can see, the menu hierarchy is represented as an outline, with sub-items indented below their parent items. This outline can represent either a menu bar or a pop-up menu. For a menu bar, the left-justified items represent the actual pull-down menus on the menu bar, while for a pop-up menu the left-justified items represent the actual menu items on the menu that initially pops up, which can have cascading submenus just as the pull-down menus on a menu bar can.

Each of the menu-items in the default menu simply prints a string to the toploop to remind you that you haven't implemented that item yet. While Allegro CL has functionality for each of these standard Windows menu bar entries, this built-in functionality is meaningful only in the context of the Lisp development environment where you are editing Lisp source code files. If you are to attach some meaning to these entries for your own application, you will need to write your own code to interpret each of these items however it is appropriate for your application.

For this tutorial, let's just add an item and test it out. Click on the **Save** item in the menu item hierarchy and then click on the **Add** button. A new menu item call New Item will be added to the hierarchy beneath the selected item, and the cursor will be placed into the Title field on the menu editor, ready for you to enter a title to be displayed on the new menu item. Enter "~Yow" in this field, then tab to the Value field and enter the following form:

```
(lambda ()(print "Yow"))
```

This creates a new menu item that prints the string "Yow". Next click on the arrow on the Synonym combo-box, and select the letter Y from the choice list, and click on the "Ctrl" check-box just below that. This will make the keyboard shortcut Control-Y invoke the new menu-item whenever the window that this menu-bar is on has the keyboard focus.

Though we're using lambda lists in this example, you probably will usually want to use symbols that name functions for your menu-item values, since (1) it's easier to find the code using find-definition on the symbol rather than looking through the large auto-generated window code file for a lambda expression, and (2) there's not much room for editing a lambda expression on the menu editor.

Also, a menu-item value is not always a function that takes zero arguments as in this example, but is instead whatever kind of value is expected by the selection function of the menu on which that menu-item lies. We just now added a menu-item under the File menu, whose selection function is **`funcall-menu-item`** (which you can see on the first line of the menu item hierarchy). **`funcall-menu-item`** is a documented Common Graphics function that expects the menu-item's value to be a function that takes no arguments, but you can use any selection function that you like instead of **`funcall-menu-item`**, provided that it takes the 3 arguments *`menu`*, *`menu-item`*, and *`stream`*.

Now that you've entered a new menu item, either press Enter or click the **Apply** button to copy the edited menu back to your dialog window, which it was read from. Since the Menu Editor is so large, it likely has covered up your dialog window at least partially; if so,

then click the **Expose** button on the Menu Editor, which exposes the window whose menu is currently being edited.

How to make a window a top-level window (so menu bar appears)

Notice that the menu bar doesn't appear yet on your window. The reason is that in MS-Windows only top-level windows can have menu bars. In Allegro CL, this means windows whose parent window is the value of `*screen*`. The interface builder initially creates all windows on `*lisp-main-window*` so that you can easily manipulate them alongside the other windows in the Lisp development environment, but you can switch any window to be top-level at any time. Pop up the window menu on your dialog window, select **Set Attribute**, then **Flags**, then **Top-Level-P**. The dialog will be recreated directly on the screen, and will have the menu bar that we created.

Even though your dialog is still in Edit mode, you can use the menu items at any time. Click on the File menu and select **Yow** (or just type Control-Y while your dialog has focus), and "Yow" will be printed in the Toploop window.

Editing top-level windows

Since your dialog is now a top-level window (its parent is `*screen*`), if you click anywhere in the main Lisp window it will cover your dialog. Editing top-level windows is when the **Expose** button on each of the IB editor dialogs is most handy for finding the window being edited. In general though, it's probably easiest to develop your windows as non-top-level ones (except while testing their menu bars), and then switch them to top-level if needed toward the end of the development period. You may not even need to do this at all when developing a runtime application, because in the runtime lisp `*lisp-main-window*` is set equal to `*screen*` since there is no main Lisp window, and so the windows that you developed on the main lisp window will be created directly on the screen at runtime anyway.

The window editor dialog

From the Builder pull-down menu, select **Window Editor**. This dialog corresponds to the Widget Editor, except that it applies to dialogs and other windows rather than to widgets. Rather than using a single list of the modifiable window attributes, the Window Editor uses separate widgets for them.

You can alternately bring up the Window Editor by selecting **Edit on Form** from an edited window's pop-up menu, or by holding the Alt key down and left-clicking a window's background. Here is the dialog.

Code to generate our dialog

We discussed saving code under the heading **How to save the auto-generated code for recreating edited windows above**. Recall that to save the source code for your dialog window, click right on its background and select **Code** from the window pop-up menu and then **Save Code to File**. A dialog appears allowing you to specify the path of the new file. Here is the code from the example dialog that we created. Notice it lacks compactness, a typical feature of machine-generated code. (This code is included to show what code generated by the IB looks like. You should not use this code yourself. Instead, generate code from the IB as we have done.)

```
;; Define the dialog :Dialog-1

(in-package :common-lisp-user)

;; Load the picture widget images from the corresponding .BML file.
(let ((pathname
         (merge-pathnames (make-pathname :type "BML")
            *load-pathname*)))
   (when (probe-file pathname) (load pathname)))
```

```
(defvar *dialog-1* nil)

;; Return the window, creating it the first time or when it's closed.
;; Use only this function if you need only one instance.
(defun dialog-1 ()
   (if (windowp *dialog-1*) *dialog-1*
      (setq *dialog-1* (make-dialog-1))))

;; Create an instance of the window.
;; Use this if you need more than one instance.
(defun make-dialog-1 ()
   (setq *loaded-but-uncreated-windows*
      (delete 'dialog-1 *loaded-but-uncreated-windows*))
   (let (window-0 window-1 window-2 window-3 window-4)
      (setq window-0
         (open-dialog
            (list
               (make-dialog-item
                  :widget 'single-item-list
                  :name :single-item-list-1
                  :title "Single Item List 1"
                  :value :three
                  :box (make-box 33 3 113 82)
                  :tabstop t
                  :groupstart nil
                  :set-value-fn 'my-set-value
                  :key 'capitalize-object
                  :range (list :one :two :three :four)
                  :font (make-font nil :arial 16 nil))
               (make-dialog-item
                  :widget 'picture-button
                  :name :picture-button-1
                  :title "statpic.bmp"
                  :value "Picture Button 1"
                  :box (make-box 134 48 159 73)
                  :border :black
                  :tabstop nil
                  :groupstart nil
                  :background-color
```

```
                (make-rgb :red 192 :green 192 :blue 192)
                :pressed-color-mapper
                (list
                  (cons (make-rgb :red 192 :green 192 :blue 192)
                        (make-rgb :red 0 :green 255 :blue 255))
                  (cons (make-rgb :red 128 :green 128 :blue 128)
                        (make-rgb :red 192 :green 192 :blue 192))
                   (cons (make-rgb :red 0 :green 0 :blue 0)
                        (make-rgb :red 128 :green 0 :blue 0)))
                :stretching t
                :cluster :default-picture-button-cluster))
           'dialog *lisp-main-window*
           :name :dialog-1
           :title "New Dialog"
           :font (make-font :swiss :system 16 '(:bold))
           :window-state :shrunk
           :window-border :frame
           :left-attachment nil
           :top-attachment nil
           :right-attachment nil
           :bottom-attachment nil
           :user-movable t
           :user-resizable t
           :user-closable t
           :user-shrinkable t
           :user-scrollable nil
           :overlapped nil
       :background-color (make-rgb :red 192 :green 192 :blue 192)
           :pop-up-p nil
           :window-interior (make-box 74 68 248 173)))
    (setf (window-editable-p window-0) t)
   (setf (getf (stream-plist window-0) :path) "z:\\tmp\\ibcode")
    (setf (getf (stream-plist window-0) :startup-state) nil)
    (setf (getf (stream-plist window-0) :top-level-p) nil)
    (setf (help-string window-0) (delete #\Newline nil))
    (setf (getf (stream-plist window-0) :package) nil)
    (setq window-1
       (open-dialog (list)
           'common-status-bar window-0
```

```
            :name nil
            :title nil
            :font (make-font :swiss :system 16 '(:bold))
            :window-state :normal
            :window-border :none
            :left-attachment :left
            :top-attachment :bottom
            :right-attachment :right
            :bottom-attachment :bottom
            :user-movable nil
            :user-resizable nil
            :user-closable nil
            :user-shrinkable nil
            :user-scrollable nil
            :overlapped nil
         :background-color (make-rgb :red 255 :green 255 :blue 255)
            :pop-up-p nil
            :window-interior (make-box 0 82 174 105)))
      (setf (window-editable-p window-1) t)
      (setf (getf (stream-plist window-1) :path) nil)
      (setf (getf (stream-plist window-1) :startup-state) nil)
      (setf (getf (stream-plist window-1) :top-level-p) nil)
      (setf (help-string window-1) (delete #\Newline nil))
      (setf (getf (stream-plist window-1) :package) nil)
      (add-common-status-bar-to-window window-0 :font
       (make-font :swiss :system 16 '(:bold)) :parts nil :min-height
          0)
       nil
     (let* ((box (getf *window-exteriors* (object-name window-0))))
         (when box (reshape-window-exterior window-0 box)))
      (show-window window-0 nil)
      window-0))
(unless (windowp *dialog-1*)
   (pushnew 'dialog-1 *loaded-but-uncreated-windows*))
```

2.2 Managing the images used by picture widgets

When dealing with picture-button and static-picture widgets programmatically, the image is stored as the dialog-item-title or dialog-item-value (respectively). The value for the image can be of several types, including the filename of a *.bmp* or *.ico* file, an icon handle, a pixmap array, or a symbol bound to either an icon handle or pixmap array. In the Builder, the following subset of types is allowed:

1. **A string naming the path of a** *.bmp* **bitmap file or a** *.ico* **icon fi**le. When a filename is used, an Allegro CL pixmap array is read initially from the specified file, and then cached internally and converted to a pixmap handle (that is, a Windows device-dependent bitmap) for faster redrawing. When you save the dialog on which the picture widget lies, the lisp pixmap array is saved along with the dialog.

   The pixmap arrays are saved in a separate file from the one that you specify for the dialog window itself. The pixmap file will be placed in the same directory and with the same filename as the usual window file, except having the extension *.bml*. The dialog window's file will contain a line at the top that will load the *.bml* file from the same directory that the dialog's file is currently being loaded from, so that you can still load everything that's needed for that dialog by simply loading its file as you do for any window. If you move the dialog's file to a different directory then just be sure to move the corresponding *.BML* along with it.

   The saved pixmap will be used from then on, so that you don't need to distribute the *.bmp* or *.ico* files with your application. If, however, you edit the original *.bmp* or *.ico* file and wish to update the widget to use the new version, you can do this by selecting the **Sync to Image File** menu item on the **Class-Dependent** submenu of the **Set Attribute** submenu of the right-button pop-up menu for the picture widget. This will remove the cache for that widget and any others that use the same image file, re-read the image from the file, and redisplay the affected widgets.

   Note that when you are entering a filename for a picture-widget, either on the Widget Editor or from the widget's right-button pop-up menu, you don't need to type quotation marks and double backslashes, even though the value is sometimes not a string.

2. **A symbol that is bound to an icon handle**. To use an actual Windows icon for a picture widget's image, you will need to call **`cg:extract-icon-from-`**

file yourself to read a *.ico* icon file and return an icon handle, and then bind a symbol to that icon handle before initially specifying that symbol as the title or value of the picture widget. You will also need to distribute the *.ico* icon file(s) with your application. To load them at runtime from the directory where *lisp.exe* is installed, you could use a form such as this:

```
(setq *icon1*
  (cg:extract-icon-from-file
    (merge-pathnames "icon1.ico"
                     allegro:*application-directory*)))
```

2.3 A few useful functions

While the Builder is an interactive tool, there are a few functions that are particularly useful to use along with it, typically in the toploop window. These functions are in the `builder` package (nicknamed `bill`) or in the `common-graphics` (nicknamed `cg`) package.

Getting an Object By Mousing On It

For all three of the functions defined under this heading:

- The *`prompt`* argument is printed in the Allegro CL status bar while waiting for an object to be clicked on.
- If the *`immediate-p`* argument is non-`nil`, then there will be no wait for you to click, and the object under the mouse cursor at the moment will be returned.

`get-widget` **[Function]**

Arguments: `&key (:prompt "Select a widget") :return-dialog-if-background-p :immediate-p :highlighting-p`

Package: `builder`

■ This function waits for you to click on a dialog-item, and then returns it or else `nil` if you click elsewhere. If *`return-dialog-if-background-p`* is non-`nil` and you click on the background of a dialog window, the dialog window will be returned rather than `nil`. If *`highlighting-p`* is non-`nil`, then all widgets on editable dialog windows will be boxed as the mouse cursor moves over them to show that they are selectable. *`prompt`* and *`immediate-p`* are defined above.

get-window **[Function]**

Arguments: `&key (:prompt "Select a window") :immediate-p :no-widgets-p :no-single-frame-child-p :screen-position`

Package: `builder`

■ Waits for you to click on any Allegro CL window, and then returns that window, or `nil` if you clicked elsewhere. If *`no-widgets-p`* is non-`nil` and a widget is clicked on, then the widget's dialog window is returned. If *`no-single-frame-child-p`* is non-`nil` and a frame-child pane window is clicked on, then the pane's parent frame window is returned. *`prompt`* and *`immediate-p`* are defined above. If *`screen-position`* is a position and *`immediate-p`* is non-`nil`, then the window at that position on the screen at that moment is returned without waiting for a click.

get-dialog **[Function]**

Arguments: `&key (:prompt "Select a dialog window") :immediate-p`

Package: `builder`

■ Waits for you to click on a dialog window, and then returns it, or `nil` if you clicked elsewhere. If a widget is clicked on, then the widget's dialog window is returned. *`prompt`* and *`immediate-p`* are defined above.

Customizing mouse behavior over objects in edit mode

edit-action **[Generic function]**

Arguments: *`window widget-or-window mouse-buttons offset`*

Package: `builder`

■ Determines what happens when a user clicks on a widget or window which is in edit mode in the Interface Builder. You can write methods on this generic function to map additional mouse button and shift key combinations to new functionality. The arguments which this generic function is called with are:

`window` -- if a widget was moused, this is the dialog on which the widget lives. Otherwise it is the window that was moused.

`widget-or-window` -- if a widget was moused, this is it. Otherwise it is the window that was moused, and is the same as the *`window`* argument.

mouse-buttons -- the **logior**'ed combination of mouse buttons such as left-mouse-button and shift keys such as ALT-KEY which were used click the window or widget.

offset -- a position denoting the distance from the upper-left corner of the widget or client area of the window at which the user clicked.

■ Example: this rather trivial example makes a Control-ALT-left-click inspect a widget or window that is clicked while editing it with the interface builder:

```
(defmethod bill:edit-action ((window basic-pane)
                             (window-or-widget t)
                             (mouse-buttons
                                (eql #.(logior control-key alt-key
                                               left-mouse-button)))
                             offset)
   (inspect window-or-widget))
```

Getting an object from its name

The following two functions return window or widget objects from their object-names. This can be handy with the Builder, since object-names are printed by default in the Allegro CL status bar as you move the mouse over windows and widgets when they are in edit mode. These are Common Graphics functions that are available in a runtime image.

window **[Function]**

Arguments: *object-name*

Package: common-graphics

■ Returns a window given its object-name, assuming that the window lies either on *lisp-main-window* or *screen*.

widget **[Function]**

Arguments: *object-name dialog*

Package: common-graphics

■ Returns a dialog-item given its *object-name* and the dialog window that it lies on. The *dialog* argument can be either the dialog window object or its object-name.

2.4 Editing a window that was not created with the builder

A particular window may be edited only if its `window-editable-p` attribute is non-`nil`. By default, this is true only of windows that have been created with the Builder. If you would like to edit a window that was created programmatically, you can do so simply by flagging it as an editable window like this:

```
(setf (window-editable-p my-window) t)
```

In order to make use of the edited version of the window, you need to stop using the original programmatic definition of it and use the one that the builder generates instead. Further, the parent of the window must be `*lisp-main-window*` or `*screen*` in order to be selectable for editing.

[This page intentionally left blank.]

Chapter 3 Menus and mouse actions

3.1 Right-Button Pop-Up Menus

Clicking the right mouse button over a window that is in edit mode pops up a menu of choices that are applicable to the item that was clicked on. If you clicked on a widget on a dialog then the Widget Menu appears, and if you clicked over the empty background of a window then the Window Menu appears.

The Window Menu is always the same except that certain items are grayed out when they are not applicable to the class of window that you clicked on. In particular, items pertaining to widgets are available only for dialog windows (and instances of dialog subclasses).

The Widget Menu is also always the same except that the **Set Attribute** submenu has a **Class-Dependent** submenu that lists all of the attributes that apply to the particular class of the widget that was clicked on but that do not apply to all widget classes.

Note that while you click the right mouse button to pop up the menu, you must release the right button and then click the left button on the desired menu item in order to select it. To select nothing from the menu, left-click somewhere off the menu.

By default, the submenus cascade to the side when you click on an item that has a submenu. This allows for perusal of the whole hierarchy of menu-items, since you can back up from each submenu to its parent, or hold the left mouse button down and drag the cursor over all of the items of a menu to see all of their submenus. On the other hand, since there are three levels of menus you sometimes have to move the mouse a long way to select the desired item. If this is annoying, you can de-select the **Use Cascading Submenus** item on the Builder Preferences dialog, which will cause the submenus to pop-up at the same loca-

tion as their parent. You can tell which mode is in effect since arrows indicate cascading submenus and ellipses (...) indicate pop-up submenus.

3.2 The Widget Pop-up Menu

Here are the choices on the right-button pop-up menu for widgets:

Object Name

The topmost item of the widget menu displays and allows you to change the object-name of the widget. The object-name is useful for finding the widget object programmatically, either by using the function **`cg:widget`** or the more general **`cg:find-named-object`**.

This menu item may appear to be a non-selectable title for the pop-up menu, since it is initially highlighted and contains the name of the object that you clicked on. And unlike the other menu items, it displays the attribute' value rather than its name. It is in fact a selectable item though.

Find Methods

Displays a menu of choices applicable to the class of the widget object.

Clone Widget

Creates a new widget of the same class as the one that you clicked on, and with the same attribute values. The new widget will appear just below the original, at which time you must move the clone where you want it and left-click to position it there. The clone will be given a unique object-name so that it can always be uniquely located programmatically using this name, but you will likely want to provide a more meaningful object name yourself.

When cloning widgets, it's usually most efficient to first edit the attributes of the widget, and then make the clone, since you typically won't need to change as many attributes of the new widget this way after cloning. For radio-buttons in particular, you can give a new cluster name to the first one, and then create clones that will be in the same cluster automatically without having to modify each clone's cluster attribute. (By default *all* radio buttons on the dialog will be in the same cluster.)

You can alternately clone a whole group of widgets at once, from the **Widget Groups** submenu of the Window Menu.

Delete Widget

Destroys the widget that you clicked on. First a dialog will pop up to prompt you for confirmation, unless you have de-selected this option on the Builder Preferences dialog. If you experience regret after deleting a widget, you can recreate it from the **Undo** submenu of the Window Menu of the dialog window that the widget was on.

You can alternately delete a whole group of widgets at once, from the **Widget Groups** submenu of the Window Menu.

Set Attribute

Pops up a submenu of modifiable attributes for the widget that you clicked on. These attributes correspond to the options for the function **`cg:make-dialog-item`** and also to the accessors with names like "dialog-item-foo" for the "foo" attribute, so you can refer to the common graphics documentation for further information about the actual attributes. The same set of attributes is also listed on the Widget Editor when you display a widget there, and you can click on particular attributes in the Widget Editor list to see help strings for them in the Allegro CL for Windows status bar.

Since there are a lot of widget attributes, they are grouped into the following further submenus:

Main Attributes --- miscellaneous frequently-modified attributes

Class-Dependent --- attributes that apply to the class of widget that you clicked on, but not to all classes of widgets

Event Handlers --- functions that you write and which are invoked when various events happen to the widget

Flags --- attributes that have only two possible states, either off or on. This submenu displays a check mark beside those items that are currently on.

Color --- attributes that control the color of the widget

Edge Constraints --- attributes that control how the widget is moved or resized when its parent dialog window is resized

Box --- attributes that control the position and size of the widget

Copy Attribute

Copies a selected attribute from the widget that you clicked on to one or more other widgets, on the same or other dialog windows. You first select an attribute from the submenu that pops up, which contains roughly the same set of attributes as the **Set Attribute** item. A rubber-band line will then stretch from the widget that you are copying from, to remind you that you are in **Copy Attributes** mode and so that you can be sure that you are copying from the correct widget. You then click on a series of other widgets in any editable dialog windows. Each widget will change just after you click on it to reflect the copied value. To stop copying, either right-click anywhere or do any click anywhere except on a widget. The rubber band line and special mouse cursor will disappear to tell that you are no longer in **Copy Attributes** mode.

You might watch for cues that each copy is successfully performed, because if you click too lightly for the click to register, or if lisp is busy garbage collecting when you click on a widget, the click can be missed. The cues include a message printed in the Allegro CL for Windows status bar while the mouse button is down and the mouse cursor changing back to the default arrow cursor while the mouse button is down, as well as any change to the appearance of the widget due to the new attribute value. If the change is not a visible one, holding the mouse down briefly while mousing each widget provides the other cues to assure you that the attribute has in fact been copied.

This submenu is always a pop-up rather than a cascading submenu so that it can qualify as a previous-action-to-repeat, allowing you to pop it up again immediately by giving the repeat-previous-operation gesture (alt-right-click or middle-click).

Reposition

Lets you change the location of the widget in certain ways. These items are most handy if you have overlapping widgets. While the Builder prevents you from achieving overlapping widgets by default, you can turn off this option on the Builder Preferences dialog.

Bury --- Moves the widget to the bottom of the occlusion stack, so that if you have overlapping widgets, this one will move behind any that overlap it. Remember that the bottommost widget will also be last in the tab order for the dialog.

Expose --- Moves the widget to the top of the occlusion stack, so that if you have overlapping widgets, this one will appear in front of any that overlap it. Remember that the topmost widget will also be first in the tab order for the dialog.

Move --- allows you to move the widget. While you can normally do this simply by left-clicking in the middle of the widget, this menu item allows you to do it even in rare instances where the middle of the widget is covered by another widget or subwindow.

Resize --- allows you to stretch the lower right corner of the widget, just as you normally would do by left-clicking its lower right corner.

Edit On Form

Selects the Widget Editor dialog and displays the widget that you clicked on, to allow you to conveniently view or edit several attributes of the widget.

Undo

Pops up a menu with a list of operations that will undo each edit that you have performed on the widget that you clicked on. The top item will undo the most recent edit, and the bottom item will undo the first edit. When you select one of the choices, that item will be removed from the widget's undo list, and a new item to undo the undoing that you just did will be added to the top of the list.

The operations that can be undone include setting attributes of the widget (either from the pop-up menus or the Widget Editor dialog) plus moving and resizing it. If you accidentally delete a widget, you can recreate it from the parent dialog window's undo list.

3.3 The Window Pop-up Menu

Here are the choices on the right-button pop-up menu for windows:

Object Name

The topmost item of the window menu displays and allows you to change the object-name of the window. The object-name is useful for finding the window object programmatically, either by using the function **`cg:window`** or the more general **`cg:find-named-object`**.

This menu item may appear to be a non-selectable title for the pop-up menu, since it is initially highlighted and contains the name of the object that you clicked on. And unlike the other menu items, it displays the attribute's value rather than its name. It is in fact a selectable menu item though.

Add Widget (Dialog Windows Only)

Allows you to create a widget on a dialog window, without bringing up the Widget Palette. The widgets are grouped into four submenus according to their basic functionality.

Widget Groups (Dialog Windows Only)

Performs operations on groups of widgets that you select.

Move Widgets --- allows you to move several widgets at once. The mouse cursor switches to an arrow with a box at its tail, to indicate that you should stretch a box around a set of widgets to indicate which ones to move. The box need only intersect with each widget rather than surrounding it. Left-click (and hold) at the upper-left corner of the region that intersects with the desired widgets, then drag out the lower-right corner of the region and release. At this time a black rectangle with the same size as the group of selected widgets will appear centered about the mouse cursor. Move the mouse to position this rectangle where you would like to move the set of widgets, and click to place them there.

Clone Widgets --- allows you to clone several widgets at once. You can think of this process as defining a class consisting of the selected widgets and instantiating that class on the fly. The selection of the widgets works as in **Move Widgets** above.

Delete Widgets --- allows you to destroy several widgets at once. A dialog window first pops up to let you confirm the deletion, unless you turn this option off in the Builder Preferences window. You are prompted only once for the entire set of widgets, with the widgets listed in the dialog so that you can make sure that you selected the correct set. You can still re-create any of the widgets individually from the **Undo** item on the dialog's right-button pop-up menu, in case you accidentally include a widget that you don't want to delete. The selection of the widgets works as in **Move Widgets** above.

Space Equally --- allows you to equalize the amount of space between successive widgets that are arranged roughly in a row or column. Select the set of widgets as in **Move Widgets** above. If the box that you stretch is taller than it is wide, then the widgets will be shifted vertically so as to make the vertical distance between successive widgets be the same; otherwise they will be shifted horizontally. The two "end" widgets remain in their current positions, and the widgets between them are moved, so you need to select at least three widgets in order to cause any movement. (Since neither "end" widget moves, inner distances may differ by one pixel due to roundoff error.)

Note that for borderless widgets such as radio-buttons and check-boxes, the widgets may not appear equidistant unless they are the same size, so unless you know they are the same size (such as if they were created by cloning) you may want to copy the box-height attribute from one to the others before spacing them equally.

Set Tab Order --- allows you to click on a sequence of widgets to establish the order in which they will be selected as the user presses the TAB key while the dialog has focus. Note that the first widget in the tab order will also be the topmost widget, and the last will be the bottommost.

A particular case where tab order is important is when you associate a static-text widget with another widget (such as a text-edit widget), and you insert a tilde (~) into the title of the static-text widget so that the user can use the alt key with the character following the tilde to focus on the widget with which the static-text widget is associated. In order for this to work, the static text widget must come immediately before the other widget in the dialog's tab order.

When you select this item, the mouse cursor changes to a "T" with an arrowhead at its base to remind you that you are in "Set Tab Order" mode. Click on the widget that you would like to come first in the tab order, and then on successive widgets. When you have clicked on all of the widgets whose tab order you care about, exit "Set Tab Order" mode by either right-clicking anywhere or left-clicking anywhere except on a widget.

While clicking on the sequence of widgets, it's important to notice that each widget is successfully selected. If you click too lightly for a click to register, or if the lisp is garbage collecting at the moment, one of the widgets may be missed. If you hold the mouse button down momentarily for each click, though, you may notice that a message is printed in the Allegro CL for Windows status bar while the button is down, and the mouse cursor changes back to the default arrow. These cues can reassure you that the current click wasn't missed. If you discover later that you did miss a widget, you can always set the "Tab Position" attribute of the widget that was missed rather than doing the whole "Set Tab Order" operation over again. To do this, set the "Tab Position" attribute of the missed widget to be the same number as the current tab position of the widget that you would like it to proceed.

Remember that a widget will not be tabbed to at all unless its **Tabstop** flag attribute is turned on.

Set Attribute

Pops up a submenu of modifiable attributes for the window that you clicked on. These attributes correspond to the options for the function **`cg:open-stream`**, so you can refer to the common graphics documentation for that function for further information about the actual attributes. Many, but not all, of these attributes are also listed on the Dialog Editor when you display a window there, and you can see help strings for them in the Allegro CL for Windows status bar by moving the mouse cursor over the Window Editor widgets for particular window attributes.

Since there are a lot of window attributes, they are grouped into the following further submenus. Note that there are no "Event Handler" attributes as there are for widgets, because events for windows are handled by CLOS methods rather than functions that are placed on individual objects as they are for widgets. So you still need to keep track of where you define your window event methods separately.

Main Attributes --- miscellaneous frequently-modified attributes

Startup State --- pops up a menu to set the state of this window to be normal, shrunk (invisible), icon, maximized, or pop-up. Pop-up is applicable only if the window is a dialog. This attribute affects only the state that the window will have when it is later recreated by the lisp code that the Builder generates for this window. It doesn't change the current state of the window since the window needs to remain in normal state during editing. That's why this attribute is called "Startup State" even though the corresponding **`cg:open-stream`** option has the different name `:window-state`.

Flags --- attributes that have only two possible states, either off or on. This submenu displays a check mark beside those items that are currently on.

Edge Constraints --- attributes that control how the window is placed with respect to its parent.

Size/Position --- attributes that modify the size or position of the window. This submenu is handy if you have turned the Resizable or Movable flag attributes off for a window and therefore can't resize or move it from its frame.

Code

Lets you manage the Lisp source code that the Builder automatically generates to programmatically recreate the window as it was created interactively with the Builder.

Code Filename --- lets you enter a new pathname to save the window's code to. You may not need to use this item at all, since when you save the window for the first time you will be prompted for a pathname if you haven't established one yet. This item is useful later, though, if you want to change the pathname where the window is saved.

Save Code to File --- generates the lisp code to recreate the window that you clicked on along with any subwindows or widgets, and saves it to a standard lisp source code file. If you have not yet established the pathname to save this window to, you are prompted for it at this time. A shortcut for this operation is to hold down the control key and left-click the background of the window while it is in edit mode. You may want to save your edited windows fairly often, just as you do the lisp code that you write in lisp-edit windows.

View Code File --- brings up a lisp-edit window containing the contents of the file where the Builder wrote the auto-generated lisp code for recreating this window and its children. You can further edit this code textually, and still be able to edit the result once again with the Builder. But remember that after editing the source code textually you must evaluate the code and then use it to recreate the window programmatically before editing the window again with the Builder, and after editing graphically with the Builder you must use the Builder to save the window to its file before editing it textually again.

Note that if the lisp-edit window for an edited window remains open while you further edit the window interactively with the Builder, and you then save the window with the Builder, and then select this **View Code** menu item to view the code, then the existing lisp-edit window will simply be exposed with the out-of-date source code. In this case you need to close the lisp-edit window and select the **View Code File** item once again to see the newly-saved code.

Reload from Code File --- destroys the window that you are editing along with its children, and recreates it by loading the lisp source code file that it is saved to and calling the function from this source code file that recreates the window. This is useful for reverting the window to the state that it and its subwindows or widgets were in when you last saved it.

Reload from Backup File --- whenever you use the Builder to save a window to its file, which is typically a *.lsp* file, the file is first copied to another file with the same name except for a *.bak* extension, and then the window is saved to the *.lsp* file. This menu item allows you to revert to the backup file, which reflects the state of the window when you did the second-most-recent save.

Miscellaneous

Various things you can do to or find about the window, including **Find Methods** (which provides choices relating to the class of the window) and the following:

Clone Window -- Creates a new window of the same class as the one that you clicked on, and with the same attribute values. Any subwindows of the window are also cloned and placed into the new window. If the window is a dialog, then each of its widgets is cloned as well. The set of cloned objects is completely independent of the original. The clone of the window that you moused will be given a unique new object-name so that it can be distinguished programmatically from the original even if you don't change the object-name yourself, but any cloned subwindows or widgets will retain their original object-names since their names are still unique with respect to their siblings.

Set Parent Window -- Allows you to create an arbitrary hierarchy of windows by clicking on a new parent window for this window. The window will be positioned in the new parent with its top left corner where you click, and then moved if needed so that it doesn't extend outside of the new parent's interior (or outside of its scrollable extent if the parent is scrollable).

If you move a window onto another editable window this way, then the only window that you need to save to file is the parent, since subwindows are saved automatically when their parent is saved.

One limitation is that you can't change the parent of a window that lies on `*screen*`, and you can't change the parent of other windows to be `*screen*`. If you need to do this, then toggle the **Top-Level-P** flag attribute of the window, which will recreate the window on `*screen*` if it wasn't already, or on `*lisp-main-window*` if it was on `*screen*`.

Delete Window -- Destroys the window that you clicked on and any subwindows or widgets. A dialog first pops up to allow you to confirm deletion, unless you turn this option off in the Builder Preferences dialog. There is no "undo" capability to recreate a deleted window.

Undo -- Pops up a menu with a list of operations that will undo each edit that you have performed on the window that you clicked on. The top item will undo the most recent edit, and the bottom item will undo the first edit. When you select one

of the choices, that item will be removed from the window's undo list, and a new item to undo the undoing that you just did will be added to the top of the list.

The operations that can be undone include setting attributes of the window plus (if this is a dialog window) recreating any widgets that you deleted from this window.

Run Window

Another way to make the window runnable rather than editable.

Edit On Form

Selects the Window Editor dialog and displays the window that you clicked on, to allow you to conveniently view or edit several attributes of the window.

Edit Menu Bar

Selects the Menu Editor dialog and displays the menu bar for the window that you clicked on, to allow you to edit the menu bar for this window. If the window has not yet been given a menu bar, a default "starter" menu hierarchy is shown on the Menu Editor that you can use as a template for creating your own menu. (If the actual menu does not appear on your window after applying your edits from the Menu Editor, the probable reason is that your window needs to have the Top-Level-P flag attribute turned on, since only top-level-windows (lying directly on `*screen*`) can have menu bars.)

3.4 Other mouse actions

In addition to the right-button menus, other mouse clicks (sometimes combined with shift keys) can be used to perform common editing tasks. The actions differ somewhat depending on whether you click a widget or a window.

By default, when you move the mouse cursor over a window or widget where the window is in edit mode, the actions for each type of click are displayed in the Allegro CL for Windows status bar as a reminder. After you become familiar with these actions, though, you may want to turn this behavior off by deselecting the **Object-Under-Mouse** item on the Builder Preferences window, because these messages cover over a lot of other informational messages that the Builder displays in the status bar after performing various operations.

Left-Click: Move or Resize Widget

When you left-click a widget, you can then either move it or stretch it in one or two directions, depending on what point on the widget the mouse cursor is over when you click. The mouse cursor changes as you move it over the widget to indicate which operation would occur at that point. Basically a move is possible in the middle of the widget, stretching in two directions simultaneously is possible near the corners of the widget, and stretching in one direction only is possible near the middle of the widget's edges.

When you left-click the background of a window, the window is simply selected and redisplayed.

Control-Left-Click: Find Widget Code, or Save Window Code

When you hold down the control key and left-click a widget, it does a find-definition on the widget's set-value-fn. This is a short cut for **Set Attribute** --> **Event Handlers** --> **Set-Value-Fn** on the widget's right-button pop-up menu. (If you group your lisp code together for all of the event handlers of a given widget, then you could effectively locate all of it with this shortcut and avoid the pop-up menu.)

When you click the background of a window instead, the source code for recreating it will be written to its file. This is a shortcut for **Code** --> **Save Code to File** on the right-button window pop-up menu.

Alt-Left-Click: Edit On Form

When you hold down the alt key and left click either a widget or a window, the special Builder dialog for editing the attributes of that object will be selected and will display that object for editing. This is a shortcut for the **Edit on Form** item of the right-button pop-up menus.

Control-Right-Click or **Middle-Click**: Repeat Previous Edit

When you either middle-click (if your mouse has three buttons) or hold down the control key and right-click a widget, the most recent edit that you performed on any widget will be repeated on the newly-clicked widget.

When you click a window's background instead, the most recent edit that you performed on any window will be repeated on the newly-clicked window.

By default, the particular action that will be repeated will be included in the Allegro CL for Windows status bar message that is displayed as you move the mouse over widgets and windows that are in edit mode.

Index

F

G

H

I

M

O

P

R

S

U

W

[This page intentionally left blank.]

Allegro CL for Windows

Foreign Function Interface

version 3.0

October, 1995

Copyright and other notices:

This is revision 2 of this manual. This manual has Franz Inc. document number D-U-00-PC0-06-51018-3-2.

Copyright © 1994, 1995 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademarks of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, Windows 95, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

Contents

[This page intentionally left blank.]

Chapter 1 Introduction and some examples

Allegro CL for Windows allows Lisp functions to call many of the Win32 API functions and any functions in C-coded dynamic link libraries (DLLs), and provides a mechanism to define callback functions in Lisp code. The functions and macros that support this capability are collectively called the *Foreign Function Interface* (FFI). The FFI defines a mapping between Lisp and C data formats and includes macros for defining C-style structures. This allows Lisp programs to build structures for C functions and to access components of C structures outside the Lisp heap.

Most symbols associated with the foreign function interface are in the `c-types` package, nicknamed `ct`.

16-bit DLL's cannot be linked to Windows 95 or NT images

When you run Lisp under Windows 95 or Windows NT, only 32-bit DLL's can be linked to the image. 16-bit DLL's are not supported. 16-bit DLL's can be linked to images running under Windows 3.1 or Windows for Workgroups.

1.1 Two examples

The first example illustrates calling a simple C function from Lisp. The second example illustrates defining Lisp callbacks. Online copies of all files can be found in the directory *ex\ffi32* included with the distribution.

Example 1: simple calls to C functions

The following file, named *tstdll32.c*, defines three simple functions (after the necessary **`LibMain`** definition):

- **`sum_ii()`**, which takes two integers and returns their sum;
- **`diff_ii()`**, which takes two integers and returns the result of subtracting the second from the first;
- **`stringchar()`**, which takes a pointer to a string and an integer.and returns the integer value of the ASCII representation of the character at the location specified by the integer.

These are, of course, fairly trivial examples. There is no reason to call a C function to add or subtract two numbers. The point is that the C function can do anything you want. All that is really being illustrated is (1) how a C function is called from Lisp, and (2) how the value returned by the C function is received by Lisp. Whatever your C function does, it must be called by Lisp, just as our simple function is called, and it will return something which Lisp may want to use, again, as happens in our example.

After the listing of *tstdll.c*, the *tstdll.def* file is listed.

```
// source file tstdll32.c
#include <windows.h>
BOOL WINAPI LibMain(HANDLE hinst, DWORD rsn, LPVOID ignore)
{
        return TRUE;
}
int WINAPI sum_ii(int i, int j)
{
        return i + j;
}
int WINAPI diff_ii(int i, int j)
{
        return i - j;
}
int WINAPI stringchar(char *sp, int i)
{
        return sp[i];
}
```

```
// end of tstdll32.c

// source file tstdll32.def
LIBRARY        tstdll32
DESCRIPTION    '32-bit test dll for acl\nt'
EXPORTS        sum_ii
               diff_ii
               stringchar
               dynamic_wrapper
// end of tstdll32.def
```

The *.c* and *.def* files are compiled and linked into *tstdll32.dll* file. We now link this DLL into Lisp and call the various functions from Lisp. Here are what we do:

1. We define a parameter that shows the location of the DLL. Note that we specify the full path of the file. If you are imitating this procedure, you would use the path of your file, which is likely different.

   ```
   (defparameter hlib "f:\\aclnt\\test\\dll\\tstdll32.dll")
   ```

2. We now define a Lisp function, **sum-ss**, that passes two integer arguments to the C function **sum_ii** and interprets the result as a short integer:

   ```
   (ct:defun-dll sum-ss ((x :short) (y :short))
      :return-type :short
      :library-name hlib
      :entry-name "sum_ii")
   ```

   We can now call **sum-ss**:

   ```
   (sum-ss 1 2) → 3
   (sum-ss 1 -2) → -1
   ```

   Bet we have to pass shorts. If we try to pass 40000 (which is not a short), we get an error:

   ```
   (sum-ss 40000 10000) → ERROR
   ```

3. **sum_ii** actually accepts 32-bit integer arguments. The Lisp call failed when passing 40000 because Lisp was told (in the call to **ct:defun-dll**) that the arguments were shorts. We can define another Lisp function that also links to **sum_ii** which accepts 32-bit integers:

```
(ct:defun-dll sum-ll ((x :long) (y :long))
   :return-type :long
   :library-name hlib
   :entry-name "sum_ii")
```

We can call **sum-ll** with larger numbers:

```
(sum-ll 32768 32768) → 65536
(sum-ll -32768 -32) → -32800
```

Why did we bother to define **sum-ss** in the first place when **sum-ll** accepts all the arguments **sum-ss** does and more? The reason is that short integers are fixnums in Lisp, while long integers can be fixnums or bignums. Bignum arithmetic is much less efficient that fixnum arithmetic. If you know a C function will be called only with short integer arguments, it is more efficient to define the corresponding Lisp function that way.

Similarly, we can link to **diff_ii()** with Lisp functions (again, we define a version which accepts shorts and a version which accepts longs)

```
(ct:defun-dll diff-ss ((x :short) (y :short))
   :call-mode :c; this is the default
   :return-type :short
   :library-name hlib
   :entry-name "diff_ii")

(ct:defun-dll diff-ll ((x :long) (y :long))
   :return-type :long
   :library-name hlib
   :entry-name "diff_ii")
```

Now we can call these functions:

```
(diff-ss 7 4); → 3
(diff-ll 60000 40000); → 20000
```

In this final example, we demonstrate passing strings to C functions. We defined the function **stringchar()**. We will define a Lisp function that passes a string and an integer to **stringchar()** and gets back the ASCII value of the character in the string position indicated by the integer.

```
(ct:defun-dll echar ((s (:char *)) (y :short))
    :return-type :unsigned-char
    :library-name hlib
    :entry-name "stringchar")
```

Here we call the new Lisp function (97 is a ASCII code for lowercase ‘a’; position 3 is after the end of the string, so the value is 0 since the C string is null-terminated):

```
(echar "abc" 0) → 97
(echar "abc" 3) → 0
```

So, all you have to do to link Lisp to a C function defined in a DLL is define a Lisp function that will call the C function. **`ct:defun-dll`** does the definition. It needs to be told the number and types of arguments that will be passed to the C function and what to expect the C function to return.

Example 2: calling back from C to Lisp

There is a not a lot to simply calling a C function and getting a return value. More complicated is calling a C function which itself calls back to Lisp. To do this, we have to define the Lisp function that will be called from C.

Her is the C code. It is in the files *dlltst32.c* and *dlltst32.def*

```
// source file for dlltst32.c
#include <windows.h>
typedef DWORD (WINAPI FAR *LISPCBPROC)(LPVOID,DWORD);
BOOL WINAPI LibMain(HANDLE hinst, DWORD dwrsn, LPVOID ignore)
{
        return TRUE;
}

int WINAPI call_callback(int (*pcb)(int,int,int),
                            int a, int b, int c)
{
      return (*pcb)(a,b,c);
}
// end of dlltst32.c

// source file dlltst32.def
```

```
LIBRARY         dlltst32
DESCRIPTION     '32-bit ffi test dll'
EXPORTS         call_callback
// end of dlltst32.def
```

First we create a variable that holds the location of the DLL. Note that the full path of the file is given. If you are trying this example, your path will likely be different.

```
(defvar tlib "f:\\aclnt\\uthunk\\dlltst32.dll")
```

Now we define a variable which will hold information placed there by the Lisp function that is called from C, and define several functions that are candidates to be called from C. The first two put information in the variable just defined. The third signals an error.

```
(defvar *cb-stack* nil)

(ct:defun-callback cb1 ((a :long) (b :long) (c :long))
 (push (list 'cb1 a b c) *cb-stack*)
 (+ a b c)); returning an integer is ok

(ct:defun-callback cb2 ((a :long) (b :long) (c :long))
 (push (list 'cb2 a b c) *cb-stack*)
 (+ a b c))

(ct:defun-callback cb3 ((a :long) (b :long) (c :long))
 (error "an error in a callback"))
```

Now we define the function, **`ccb`**, that will call the C function (that in turn will do the callback).

```
(ct:defun-dll ccb ((pf (:void *)) (a :long) (b :long) (c :long))
 :call-mode :c
 :return-type :long
 :library-name tlib
 :entry-name "call_callback")
```

Finally, we define the function **`c`**, which selects the function that will be called back to, and then calls **`ccb`**.

```
(defun c (n i j k)
     (ccb (ct:get-callback-procinst n) i j k))
```

We initialize `*cb-stack*` and try some calls.

```
(setq *cb-stack* nil)

(c 'cb1 1 2 3) → 6
(c 'cb2 4 5 6) → 15
(c 'cb3 7 8 9) → INVOKES AN ERROR
```

1.2 Accessing a C string from Lisp

It is not possible to return a string from a C function which is called from Lisp. C and Lisp represent strings slightly differently. In this section, we describe how to call a C function one of whose arguments is a pointer to a string. Assuming that the C function fills the string, we show how Lisp can access the filled string.

To call a C function that takes a pointer to a string as one of its arguments and then fills in the contents of the string, use **`ccallocate`** to create a character vector of the maximum required size, as in (this is for strings up to 256 characters in length):

```
(setq foo (ct:ccallocate (ct:char 256))
```

Pass this value as the *`string`* argument to the C function. Once the C function has returned, you can convert the string to a proper Lisp string object with:

```
(subseq foo 0 (ct:strlen foo))
```

Here **`ct:strlen`** returns the length of the null-terminated character vector as filled in by the C function. A new Lisp string is returned by the call to **`subseq`**.

Alternately, you can reference individual elements of the returned string without consing a new Lisp string by using **`ct:cref`**:

```
(cref (ct:char *) foo 3)
```

The above form will return the integer contained at byte 3 of the vector. You can use **`int-char`** to convert it to an actual Lisp character if desired.

[This page intentionally left blank.]

Chapter 2 FFI functionality

The macros, functions, and variables in the FFI are listed here and described in more detail in chapter 3.

2.1 Defining DLL's to Lisp

ct:defun-dll	macro	defines a foreign function to be called from lisp
ct:list-dll-libraries	function	returns list of referenced DLL libraries
ct:rename-dll-libraries	function	changes names of DLL libraries
ct:unlink-dll-functions	function	forces reloading of DLLs
ct:unlink-dll	function	forces reloading of a specific DLL
ct:dll-handle	function	returns the library handle of a DLL

2.2 Defining lisp functions to process callbacks

ct:defun-callback	macro	defines a c-callable lisp function
ct:default-callback-style	variable	argument style for callback (C or pascal)
ct:defun-c-callback	macro	overrides default-callback-style
ct:defun-pascal-callback	macro	overrides default-callback-style
ct:get-callback-procinst	function	returns a pointer C can use to call the function

2.3 Mapping C data types and structures

ct:defctype	macro	defines a name for a C type
ct:defcstruct	macro	describes a C structure type
ct:defshandle	macro	defines a new short handle type
ct:deflhandle	macro	defines a new long handle type
ct:*export-c-names*	variable	controls auto export of ctype names
ct:callocate	macro	allocates C data in the heap at runtime
ct:ccallocate	macro	allocates C data in the heap at compile time
ct:cref	macro	access field of C-type object
ct:cset	macro	set field of a C-type object
ct:csets	macro	set multiple fields of a C-type object
ct:far-peek	function	copies data from a foreign location to a Lisp location.
ct:far-poke	function	copies data from a Lisp location to a foreign location.

ct:hnull	variable	a long cpointer with null C value
ct:sizeof	macro	length of a C type
ct:strlen	function	length of heap-allocated C string
ct:handle-value	macro	get or set the handle value of a lisp handle
ct:handle=	macro	compare two handles to see if they have equal values
ct:null-handle	macro	expands to a null handle object
ct:null-handle-p	macro	test if a handle is a C null handle
ctcpointer-value	macro	value of cpointer as an integer
ct:null-cpointer-p	macro	test if a cpointer is a C null
:void	keyword	builtin C type
:char	keyword	builtin C type
:unsigned-char	keyword	builtin C type
:short	keyword	builtin C type
:unsigned-short	keyword	builtin C type
:long	keyword	builtin C type
:unsigned-long	keyword	builtin C type
:short-bool	keyword	builtin C type
:long-bool	keyword	builtin C type
:single-float	keyword	builtin C type
:double-float	keyword	builtin C type
:short-handle	keyword	builtin C type
:long-handle	keyword	builtin C type

2.4 Different views of data

In C, data is what the program finds in a storage area. The storage area can be seen as a place that holds a string of bits of some fixed size. The bit string can be interpreted as a sequence of fields of varying types and sizes (characters, signed or unsigned integers,

floats, or pointers to other storage areas) depending on the type declarations and casts in the C program. By using explicit type casts the program can interpret any data in any way the programmer desires.

Data items are found in storage in Lisp, as well. But Lisp items are best thought of as objects. Each has a header that defines its type unambiguously, and the opportunities for overriding Lisp's idea of an object's type are severely limited. Furthermore, Lisp's memory management may move an object from one area of memory to another at any time, adjusting all references to that object to point to the new location.

When Lisp calls a C function that expects a long integer argument and returns a float result, the FFI must convert the supplied argument from Lisp format to C format, push the converted value onto the stack and transfer control to the C function's entry point. When the C function returns, the FFI must convert the return value from C format to Lisp format. Lisp data is always tagged with its type, so a Lisp function definition doesn't need to include type declarations for the arguments and return value. C data is not tagged, so function definitions need type declarations.

2.5 32-bit and 16-bit DLL's

Allegro CL can call functions in 32-bit (Win32) DLL's when running under any version of Windows. It can call functions in 16-bit (Windows 3.1) DLL's *only* **when running under Windows 3.1 or Windows for Workgroups**. The remainder of this section applies only to the latter case.

When 16-bit DLL's are supported, the adjustment is handled by using the `:16-bit` keyword in the **`defun-dll`** macro and setting `*default-callback-style*` to the correct value. Two areas of concern for the programmer, however, are pointers and handles.

Everything works smoothly when calling a 32-bit DLL. The DLL is mapped into Lisp's address space and pointers are 32-bit addresses in this space. The DLL and Lisp both use the 32-bit API to allocate and transmit handles, so there is no confusion as to their meaning.

16-bit DLL's, however, use pointers in the 16-bit segment-offset form. Passing the address of an object in the Lisp heap as an argument requires converting the 32-bit address to a 16:16 pointer that maps to the same virtual memory. The FFI handles this, but only if it knows the argument is a pointer. It cannot now perform the mapping on fields within a structure when the structure is to be passed as an argument.

Chapter 3 Reference guide

This chapter gives the formal definitions of the functions, macros, etc. used by the foreign function interface. Most symbols naming objects in the foreign function interface are in the `c-types` package, nicknamed `ct`.

3.1 C Type Specifications

When supplying data to and using data from foreign functions we use Lisp symbols and lists called *c-type-specs* to describe the format of the data. Each c-type-spec describes both the data layout and interpretation for the foreign function and the Lisp representation of that foreign data.

A c-type-spec can be defined recursively as we now describe. First, the table shows the primitive building blocks and then we describe how compound specs can be created.

c-type-spec	Description	Corresponding Lisp value
`:void`	appears in (:void *) or to specify no return from a C function	`nil`
`:char`	8-bit signed integer value	Lisp integer
`:unsigned-char`	8-bit unsigned integer value	Lisp integer
`:short`	16-bit signed integer value	Lisp integer
`:unsigned-short`	16-bit unsigned integer value	Lisp integer
`:long`	32-bit signed integer value	Lisp integer

c-type-spec	Description	Corresponding Lisp value
`:unsigned-long`	32-bit unsigned integer value	Lisp integer
`:short-bool`	16-bit value treated as false if and only if it is 0	`nil` or `t` (0 maps to `nil`)
`:long-bool`	16-bit value treated as false if and only if it is 0	`nil` or `t` (0 maps to `nil`)
`:single-float`	32-bit single-float value	Lisp float value
`:double-float`	64-bit double-float value	Lisp float value
`:short-handle`	16-bit handle and 8-bit tag (for 16-bit DLL's) [16-bit DLL's are not supported when running under Windows 95 or Windows NT.]	`short-handle` (a distinct Lisp data type)
`:long-handle`	32-bit handle with a 16-bit type tag	`long-handle` (a distinct Lisp data type)

Plus, any type defined by **`defctype`**, **`defcstruct`**, **`defshandle`**, or **`deflhandle`**. (All these macros are defined below.)

The following compound types are also c-type-specs:

(*`c-type-spec integer`*)

A vector of *`integer`* occurrences of *`c-type-spec`*

(*`c-type-spec *`*)

A 32-bit near pointer to an occurrence of *`c-type-spec`*

(*`c-type-spec **`*)

A 16:16 far pointer to an occurrence of *`c-type-spec`*

(*`c-type-spec mod1 mod2 ... modn`*)

where each *`modi`* is an integer, `*`, or `**`. This is equivalent to

```
(... ((c-type-spec mod1) mod2) ... modn)
```

These map to Lisp strings or cpointers, depending on the context.

Examples

`(:char *)` is a 32-bit near pointer to an 8-bit byte, such as is appropriate to describe a string argument.

`(:short 10)` is a 20-byte vector of 10 signed shorts.

`(:long-bool * 2)` is an 8-byte vector of 2 32-bit near pointers to 32-bit C booleans.

3.2 Functions, macros, variables, etc.

The definitions are in alphabetical order.

`callocate` [Macro]

Arguments: *`c-type-spec`* `&rest` *`inits`*

Package: `c-types`

`ccallocate` [Macro]

Arguments: *`c-type-spec`* `&rest` *`inits`*

Package: `c-types`

■ These macros expand into forms that produce instances of c-type objects. **`callocate`** produces code that allocates a new instance each time the code is evaluated. **`ccallocate`** produces code that allocates the instance once at compile time, effectively producing a quoted constant in the code (thus *c*onstant *callocate*). The *`inits`* are alternating slotnames and values for initializing components of the object. For **`callocate`** the initializations are performed each time the code is evaluated. For **`ccallocate`** they are performed once, at compile time. In either case the objects are allocated as Lisp objects in the Lisp heap.

■ *`c-type-spec`* is the c-type-spec of the desired object.

■ The *`inits`* argument should be alternating field names and values. If *`c-type-spec`* names a C structure then the structure's field names are all acceptable. Two pseudo fieldnames, `:size` and `:initial-value`, are sometimes allowed. `:size` is used with vectors and cstructures to specify a size determined at execution time. `:initial-value` is used with cpointers and handles to set the value.

cpointer-value [Function]

Arguments: *cpointer*

Package: c-types

■ This function returns the 32-bit pointer contained in the cpointer object. The value is returned as an integer. This form is **setf**able. *cpointer* can be a 16:16 or a 32-bit cpointer.

cref [Macro]

Arguments: *ctype object access* &optional *value-object value-type alt-value-object*

Package: c-types

■ This macro extracts and returns a field value or address from a cstructure or array. **cref** is **setf**able. The arguments are as follows:

Argument	Description
ctype	The ctype of the object from which a field is to be extracted. This must be a pointer, array, or cstructure type.
object	A form evaluating to an object compatible with ctype. Compatibility is defined in the table below this table.

Argument	Description
access	Can be: • a symbol naming a field of the structure ctype, • an integer selecting an element of a vector ctype • the symbol `*` dereferencing a cpointer ctype • the symbol `ct:&` selecting the address of the ctype data, • the symbol `nil` selecting the entire ctype object, • or a list whose elements provide, from right to left, a selection sequence from object to the desired element. Each element can be any of the above or a list of the form (fixnum form) or (integer form), except that `ct:&`, if it appears, must be last. Example: `(a * b (fixnum i) ct:&)` would say that starting with a cstructure, grab the contents of the field named `a`, a pointer, dereference the pointer to get to another cstructure, select that structure's `b` field to get a vector of whatever, select the i-th element of that vector, return the address of that element.
value-object	If this is omitted, the macro expansion allocates an object of the appropriate type and destructively overlays it with the data selected by the access path. If a *value-object* form is included in the **cref** macro it must evaluate to an object of the correct type. In either case the **cref** form returns this object or *alt-value-object* or an equivalent fixnum.
value-type	This can be coded as a ctype compatable with the type of the selected field as determined from *ctype* and *access*. It can also be one of `character` or **string**. The former causes an integer field to be converted to a character by applying **int-char**. The latter causes characters to be copied from the selected field to the destination object up to the first null.

Argument	Description
alt-value-object	This should be omitted unless the field being extracted is defined as an `:unsigned-long`. Then value-object must be omitted or coded as `(make-bignum 2)` and *`alt-value-object`* must be omitted or coded as `(make-bignum 4)`.

The following table shows the Lisp objects compatible with various ctypes. The value of the *`object`* argument should be a form that evaluates to a Lisp object compatible to the *`ctype`* argument.

ctype	Compatible Lisp objects
`cstructure-type`	• a Lisp cstructure with matching tag, • a Lisp cpointer to such a cstructure, • a Lisp string at least as long as the cstructure
`(cstructure-type *)`	a Lisp cstructure with a matching tag
`(cstructure-type **)`	• a Lisp cpointer to such a structure, • a Lisp string at least as long as cstructure

`cset` [Macro]

Arguments: *`ctype object access value-object`* `&optional` *`value-type`*

Package: `c-types`

■ Evaluating this macro stores a value into a field of the ctype object. The arguments are as follows:

Argument	Description
`ctype`	A c-type-spec, as for **`cref`**.

Argument	Description
object	A form evaluating to an object compatible with *ctype*. Compatibility is as for **cref** above.
access	An access item or list, as for **cref** except ct:& is not permitted.
value-object	A form that evaluates to the object whose value is to be stored in the selected field of *object*.
value-type	Can be character or string. The former causes the conversion of a character-values *value-object* via **char-int**. The latter causes characters to be copied up to a null byte.

csets [Macro]

Arguments: *ctype object* &rest *accesses-and-values*

Package: c-types

- This macro is a multi-field version of **cset**. The arguments are as follows:

Argument	Description
ctype	A c-type-spec, as for **cset**.
object	A form evaluating to an object compatible with *ctype*. Compatibility is as for **cref** above.
access-and-values	a sequence of alternating access specifications and value-object forms. The value forms are evaluated and assigned to the designated fields in sequence.

default-callback-style [Variable]

Package: c-types

■ The value of this global variable determines the type of callback constructed by **`defun-callback`**. Legal values are

`:c` -- 32-bit caller using C calling conventions

`:pascal` --32-bit caller using Pascal calling conventions

`:c-16` -- 16-bit caller using C calling conventions

`:pascal-16` -- 16-bit caller using Pascal calling conventions

■ 16-bit DLL's are not supported when running under Windows 95 or NT.

`defcstruct` [Macro]

Arguments: *`name-and-props field-specs`* `&optional` *`tag`*

Package: `c-types`

■ Evaluating this macro defines a new structure c-type, naming and describing its fields. The name becomes a valid c-type-spec. The fields can be used with **`cref`**, **`cset`**, and **`csets`**. Objects of the new cstruct type allocated in the Lisp heap are tagged to support argument validation. The arguments are as follows:

Argument	Description
`name-and-props`	Either a symbol, naming the cstructure type, or a list of the form (*`name`* `:pack` *`alignment`*) where *`name`* is the symbol naming the structure type and *`alignment`* is either 1, 2, or 4. Leaving alignment unspecified is the same as specifying it as 1. The value used for alignment affects the insertion of padding between fields of certain types.

Argument	Description
field-specs	A non-empty list of field-specs. Each field-spec is either a simple field definition or a union definition. A simple field definition defines one field and is a list of the form (*fieldname c-type-spec*) or (*fieldname . c-type-spec*). The former is convenient for simple c-type-specs like :short or **win:hwnd**, as in (subspec :short). The latter format is convenient when the c-type-spec includes pointers or indexing, as in (filename :char 8). A union definition is a list of the form (:union field-spec-1 ... field-spec-n) and represents a c-style union of the fields defined by the *field-spec-i*.
tag	If present this must be a non-negative integer to be used as the tag for this cstructure type. If this item is omitted from the **defcstruct** form then the system will select a tag value at the time the **defcstruct** is expanded.

defctype [Macro]

Arguments: *name c-type-spec*

Package: c-types

■ This macro defines a new name for any valid c-type-spec. The arguments are as follows:

Argument	Description
name	the new name to be associated with the c-type. After the defctype is evaluated name becomes a synonym for *c-type-spec*.
c-type-spec	Any c-type-spec.

deflhandle [Macro]

Arguments: *name* &optional *tag*
Package: c-types

defshandle [Macro]

Arguments: *name* &optional *tag*
Package: c-types

■ Evaluating these macros defines *name* as a c-type-spec denoting a tagged handle. **deflhandle** defines a long (Win32) handle with a 32-bit handle-value. **defshandle** defines a short (Windows 3.1) handle with a 16-bit value. [16-bit DLL's are not supported when running under Windows 95 or NT.] All handles of this type will be tagged. The tag associated with the type is *tag*, if that is present, otherwise it is chosen by the system. The arguments are as follows:

Argument	Description
name	The name for the new handle type. This becomes a new c-type-spec.
tag	If present, this must be a non-negative integer.

defun-callback [Macro]

Arguments: *name arg-list-and-types* &rest *body*
Package: c-types

defun-c-callback [Macro]

Arguments: *name arg-list-and-types* &rest *body*
Package: c-types

■ These macros define Lisp functions that are to be invoked from outside Lisp as callbacks. Once such a function has been defined **get-callback-procinst** must be used to get an address that a foreign function can use to call it. Lisp must be told whether the calling function will be using the C or Pascal calling conventions, and whether it will be running in 16-bit or 32-bit mode. [16-bit DLL's are not supported when running under Windows 95 or NT.] Callbacks for the builtin Win32 API

functions will get control via C conventions from 32-bit code. Callbacks to be used by 16-bit DLL's may be either Pascal or C and will, of course, be called from 16-bit code. [16-bit DLL's are not supported when running under Windows 95 or NT.] **`defun-c-callback`** always defines a callback for 32-bit mode. **`defun-callback`** depends on the value of the symbol `default-callback-style` at the time the **`defun-callback`** is expanded to specify the calling environment.

■ The arguments are as follows:

Argument	Description
`name`	a symbol naming the function. **`get-callback-procinst`** will use this symbol to allocate a c-callable address that will invoke the function.
`arg-list-and-types`	a list of the form (`(`*`symbol c-type-spec`*`) ...`) with one entry for each argument passed from the foreign function. The *`symbol`* serves as a name for the argument and the *`c-type-spec`* specifies the type of argument being passed by the foreign function. The effects of different type specifications are described in a table below.
`body`	Lisp forms that make up the body of the function. The arguments can be referenced within body using the names defined in *`arg-list-and-types`*. The value of the last form in body determines the value returned to the foreign caller. If this is an integer it is returned as a long. If it is a cpointer then the cpointer-value is extracted and returned. Any other data type results in a return value of 0.

■ The following table shows the allowable types in the argument list. These can be the second elements of the lists that make up the list that is the value of *arg-list-and-types*. [16-bit DLL's are not supported when running under Windows 95 or NT.]

c-type-spec	C argument size (*16-bit* arg is nil) /(*16-bit* arg is non-nil)	arg converted to Lisp type
:char	32/16	(integer -128 127)
:unsigned-char	32/16	(integer 0 255)
:short	32/16	(integer #x-8000 #x7fff)
:unsigned-short	32/16	(integer 0 #xffff)
:long	32/32	(integer #x-80000000 #x7fffffff)
:unsigned-long	32/32	(integer 0 #xffffffff)
:long-bool	32/32	0 -> nil non-zero -> non-nil
:short-bool	32/16	0 -> nil non-zero -> non-nil
any handle type	32/16	a handle of the specified tag
:single-float	32/32	float
:double-float	64/64	float
(ctype *) or (ctype size)	32/32	32-bit cpointer (when *16-bit* is nil) 16:16 cpointer (when *16-bit* is non-nil)

defun-dll [Macro]

Arguments: *lisp-name arg-list-and-types* &key return-mode return-type library-name entry-name 16-bit call-mode

Package: c-types

■ This macro defines a Lisp function that converts its arguments for passing to a foreign function, invokes the specified foreign function, and converts the foreign function's result back to Lisp format. The arguments are as follows:

Argument	Description
lisp-name	name of the Lisp function being defined
arg-list-and-types	a list of the form ((*symbol c-type-spec*) ...) with one entry for each argument passed to the foreign function. The symbol serves as a name for the argument and the c-type-spec specifies the type of argument expected by the foreign function. The effects of different type specifications are described in a table below. The lisp function will take a fixed number of arguments, either the number of elements in *arg-list-and-types* or one more than that number, depending on the value specified for *return-mode*.

Argument	Description
return-mode	one of `nil`, `:dynamic`, `:static`, or `:smash`. `nil` is the default. Specifying `nil` or `:dynamic` produces a lisp function that allocates a new return-type object on each call and returns that object (or an equal fixnum). Specifying `:static` produces a lisp function that has a single return-type object that is destructively overwritten with the foreign function's return value. The same object is used each time the lisp function is called. The lisp function returns either that object or a fixnum of equal value. Specifying `:smash` means that the lisp function takes one more argument than the foreign function. This extra argument, the last argument in the call, must be an object of a type appropriate to be modified to hold the result from the foreign function. That is, it must in general be an object of the type specified by *`return-type`*, but see the special treatment of `:unsigned-long` in the *`return-type`* description below.

Argument	Description
return-type	a c-type-spec specifying the type returned by the foreign function. This will be mapped to a Lisp type as follows: `:void` - `nil` `:char` - a fixnum `:unsigned-char` - a fixnum `:short` - a fixnum `:unsigned-short` - a fixnum or a bignum; for `:smash` mode, a bignum must be supplied. The Lisp function will return a fixnum or a bignum as appropriate. `:long` - a fixnum or a bignum; for `:smash` mode, a bignum must be supplied. The Lisp function will return a fixnum or a bignum as appropriate. `:unsigned-long` - a fixnum or a bignum; for `:smash` mode, a vector of two bignums must be supplied - as built by `(vector (make-bignum 2) (make-bignum 4))`. The Lisp function will return a fixnum or one of the bignums as appropriate. `:short-bool` - the Lisp function returns `nil` if the foreign function returns a short 0, returns `t` otherwise. `:long-bool` - he Lisp function returns `nil` if the foreign function returns a long 0, returns `t` otherwise. `:single-float` - a lisp float `:double-float` - a lisp float `:short-handle` - a tagged short-handle `:long-handle` - tagged long-handle `(c-type-spec *)` - a 32-bit near cpointer `(c-type-spec **)` - a 16:16 far pointer

Argument	Description
library-name	A form that will evaluate to a file namestring or pathname for the DLL containing the foreign function to be invoked by the new Lisp function. This form will be evaluated when the **defun-dll** form or its compiled equivalent is loaded. Allegro CL will not actually try to load the DLL until the first time a call is made to a function in that DLL, at which time it will link the Lisp function to the C function. It will be necessary to break these links if an image is saved for a later Lisp session, since the new Lisp will not have loaded the DLL. See **rename-dll-libraries**, **unlink-dll-functions**, and **unlink-dll.**
entry-name	A string naming the foreign function's entry point as exported from the DLL.
16-bit	True if the DLL is a 16-bit Windows 3.1 DLL, otherwise `nil`. The default is `nil`. [16-bit DLL's are not supported when running under Windows 95 or NT.]
call-mode	`:c` or `:pascal`. This specifies the argument order and who clears the stack. `:c` is the default and is appropriate for 32-bit DLL's. For 16-bit DLL's, either mode can occur, and the correct mode for Windows 3.1 API functions is `:pascal`.

■ The following table shows how arguments are passed to foreign functions. Note that the last column applies only to images running under Windows 3.1 or Windows for Workgroups because 16-bit DLL's are not supported when running under Windows 95 or NT. The c-type-specs are the second element of the lists that make up the list that is the value of *arg-list-and-types*. An error will be signaled if an

argument of any type other than those listed is supplied..

c-type-spec	lisp-type	converted to (*16-bit* arg is nil)	converted to (*16-bit* arg is non-nil)
`:char`	`(integer -128 127)` `character`	32-bit signed binary 32-bit ascii value of char	16-bit signed binary 16-bit ascii value of char
`:unsigned-char`	`(integer 0 255)` `character`	32-bit unsigned binary 32-bit ascii value of char	16-bit unsigned binary 16-bit ascii value of char
`:short`	`(integer #x-8000 #x7fff)`	32-bit signed binary	16-bit signed binary
`:unsigned-short`	`(integer 0 #xffff)`	32-bit unsigned binary	16-bit unsigned binary
`:long`	`(integer #x-80000000 #x7fffffff)`	32-bit signed binary	32-bit signed binary
`:unsigned-long`	`(integer 0 #xffffffff)`	32-bit unsigned binary	32-bit unsigned binary
`:long-bool`	`nil` `anything else`	32-bit 0 32-bit non-zero value	32-bit 0 32-bit non-zero value
`:short-bool`	`nil` `anything else`	32-bit 0 32-bit non-zero value	16-bit 0 16-bit non-zero value
`any handle type`	a handle with a matching tag (two tags match if they are eql or if either is 0)	32-bit handle value	16-bit handle value
`:single-float`	`float`	32-bit float	32-bit float
`:double-float`	`float`	64-bit double	64-bit float

c-type-spec	lisp-type	converted to (*16-bit* arg is nil)	converted to (*16-bit* arg is non-nil)
(ctype *) or (ctype size)	ctype 32-bit cpointer 16:16 cpointer string	32-bit address of object's data portion 32-bit cpointer-value [error] 32-bit address of string	16:16 address of object's value 16:16 equivalent of cpointer-value cpointer-value unchanged 16:16 address of string's data

dll-handle [Function]

Arguments: *library-name*

Package: c-types

■ This function returns the windows or NT handle of the named library if it has been loaded, nil if it hasn't. *library-name* should be a string, symbol, or pathname naming the DLL

export-c-names [Variable]

Package: c-types

■ The value of this variable controls the exporting of names of c types, cstructs and cstruct fields. If *export-c-names* is true when the defining form is expanded, then the names are exported when the defining form is evaluated. Otherwise they are not exported. The initial value of this global variable is nil. It is almost always appropriate for it to be nil.

far-peek [Function]

Arguments: *pdest.lisp psource.c offset length*

Package: c-types

far-poke [Function]

Arguments: *psource.lisp pdest.c offset length*

Package: c-types

■ **far-peek** copies *length* bytes to the beginning of *pdest.lisp* from ((char *) *psource.c*) + *offset*.

far-peek returns *pdest.lisp*.

■ **far-poke** copies *length* bytes from the beginning of *psource.lisp* to ((char *) *pdest.c*) + *offset*.

far-poke returns *psource.lisp*.

■ The arguments are as follows:

Argument	Description
pdest.lisp (for **far-peek**) *psource.lisp* (for **far-poke**)	any Lisp object which can be used as a 0:32 near pointer; generally a string or cstructure.
psource.c (for **far-peek**) *pdest.c* (for **far-poke**)	a 16:16 or 0:32 cpointer.
offset	an integer indicating the offset to data relative to *psource.c* (for **far-peek**) or *pdest.c* (for **far-poke**)
length	an integer indicating the number of bytes to copy.

get-callback-procinst [Function]

Arguments: *callback-function-name*

Package: c-types

■ This function returns a cpointer whose cpointer-value can be used as an entry point from foreign code to call the named callback function. The cpointer will be a 32-bit near pointer if the callback was defined as from 32-bit code, a 16:16 far pointer if the callback was defined as from 16-bit code. *callback-function-name* should be a symbol naming the callback function. This must be the symbol specified as the name in a **defun-callback** or **defun-c-callback** macro.

handle-value [Macro]

Arguments: *handle-type handle-form*

Package: c-types

■ This macro expands into code to evaluate *handle-form* and access the foreign handle code embedded in the resulting Lisp handle object, which must be of the specified type. This form is **setf**able. *handle-type* should be a symbol naming a handle type. *handle-form* should be a form that evaluates to a handle of the same size as the specified type.

handle= [Macro]

Arguments: *handle-type form1 form2*

Package: c-types

■ This macro expands into code that compares the values, but not the type codes, of two handles. *handle-type* must be a symbol that has been defined to name a handle. *form1* and *form2* should be forms that evaluate to handles. **handle=** returns true is the handle-values of the two handles are numerically equal; otherwise **handle=** returns nil.

equal and **equalp**, two standard Common Lisp functions for comparing objects that have components, can also be used to compare handles.

(equal *obj1 obj2*) and (equalp *obj1 obj2*), when *obj1* evaluates to a handle, return true if and only if:

- *obj2* evaluates to a handle of the same length (short or long) as *obj1*, and
- the handle-value of *obj1* is eql to the handle-value of *obj2*; and
- either the handle-type of *obj1* is **eql** to that of *obj2* or either *obj1* or *obj2* has handle-type 0 (which is the wild type).

Thus, using equal or equalp differs from handle= in that the handle-type of the handles is compared.

equal and **equalp** can also be used to compare cpointers.

(equal *obj1 obj2*) and (equalp *obj1 obj2*), when *obj1* evaluates to a cpointer, return true if and only if:

- *obj2* evaluates to a cpointer, and
- the cpointer-value of *obj1* is eql to the cpointer-value of *obj2*; and

- either the cpointer-type of *obj1* is **eql** to that of *obj2* or either *obj1* or *obj2* has handle-type 0 (which is the `(:void *)` type).

`ct:hnull` **[Variable]**

■ The value of this global variable is a 32-bit near cpointer corresponding to a `(void *)0`. This value is appropriate to pass as a null value for foreign functions that expect a pointer argument.

■ In earlier releases, this variable was named by the symbol `ct:null`. However, there is also a symbol `common-lisp:null` (indeed `ct:null` and `common-lisp:null` named the same symbol). We have changed the variable name to `ct:hnull` to avoid overloading the Common Lisp symbol. However, `common-lisp:null` retains its meaning as a synonym for `ct:hnull` for Release 3.0 to preserve backward compatibility. Please use `ct:hnull` in all new code and rewrite previously written code as convenient.

`list-dll-libraries` **[Function]**

Arguments: *all*

Package: `c-types`

■ This function returns a list of DLL names referenced by the current Lisp image. If *all* is `nil` then only those DLL's that have been loaded are included in the list. If *all* is not `nil` then all DLL's that have been referenced in foreign function definitions are included, whether loaded or not.

`null` **[Variable]**

Package: `common-lisp`

■ See the description of `ct:hnull` just above.

`null-cpointer-p` **[Macro]**

Arguments: *cpointer*

Package: `c-types`

■ This macro expands into code to evaluate *cpointer* and test the resulting cpointer's value. If the cpointer-value is 0, indicating a null pointer, the code returns true, otherwise it returns `nil`. *cpointer* should be a form evaluating to a cpointer.

null-handle [Macro]

Arguments: *handle-type*

Package: c-types

■ This macro expands into a constant handle of the specified type and 0 handle value. handle-type should be a symbol naming a handle type.

null-handle-p [Macro]

Arguments: *handle-type handle*

Package: c-types

■ This macro expands into code to evaluate handle and test the resulting handle. The code returns true if the handle's value is 0, `nil` otherwise.

rename-dll-libraries [Function]

Arguments: *mapping-alist*

Package: c-types

■ This function searches through the foreign function definitions and matches the library associated with each one against the *mapping-alist*. The match is done by **eql**. Each time a match is found, the foreign function definition is altered so that the new library name replaces the old one in the definition. If no match is found in the alist then the definition is left unchanged.

mapping-alist should be an association list mapping old library names to their replacements. Each entry in the alist is of the form (*old-name* . *new-name*) where both *old-name* and *new-name* are keyword symbols whose print-names are strings naming the DLL files.

sizeof [Macro]

Arguments: *c-type-spec*

Package: c-types

■ This macro expands into the size of an object of the specified type. This size is computed in C terms, and thus it does not include the Lisp header for an object of that type allocated in the heap. *c-type-spec* must be a valid name for a c-type at macro expansion time.

strlen [Function]

Arguments: *string*

Package: c-types

■ This function returns the number of characters in *string* before the first null byte. This corresponds to the string's length from C's point of view. *string* must be a simple string of length less than 32768 characters.

unlink-dll [Function]

Arguments: *library-name*

Package: c-types

■ This function finds all foreign functions defined as being loaded from the named DLL and marks them to reestablish the linkage the next time each is called. The DLL is then unloaded. Calling this function is appropriate when a DLL has been recompiled while the Lisp image is running. *library-name* must be a string, symbol or pathname naming the DLL file to be unloaded.

unlink-dll-functions [Function]

Arguments:

Package: c-types

■ This function marks all foreign function definitions so that each will reestablish its link with the foreign function the next time it is called. A call to this function is part of the standard system initialization sequence since connection must be reestablished with any DLL's that might have been loaded when the image was saved. It can also be used to force reloading of any DLL's that have been rebuilt while the Lisp image was running.

[This page intentionally left blank.]

Chapter 4 DDE interface

DDE stands for Dynamic Data Exchange. Quoting from the *Windows API Bible*[1] (a standard reference for programming with Windows):

> Dynamic Data Exchange is a message protocol that allows Windows applications to exchange data.

In this chapter, we describe the functionality available in Lisp for DDE. Note that We do not describe DDE in much detail. We assume you are familiar with using DDE between Windows applications. If you are not, please refer to standard Windows programming manuals.

Creation and management of DDE "string handles" is done automatically. You simply pass in Lisp strings or keyword symbols that name applications, topics, and items without worrying about the string handles.

Rather than deal with the DDE callback function directly, you can simply supply a few Lisp methods which are invoked automatically when callbacks occur.

Conversations with other tasks are implemented as CLOS instances (called *ports*), so you don't need to deal with the conversation handles directly.

Lisp can act as either a client or server, and handle cold, warm, and hot links. Currently the only clipboard data format supported is text.

The functions and variables documented below are all exported from the `dde` package.

Example

There is a simple example illustrating the DDE functionality in *ex\dde\examples.lsp*. Also in that directory is ddedoc.txt which contains most of the information in this chapter.

1. *The Waite Group's Windows API Bible*, Mill Valley, California, Waite Group Press, 1992.

4.1 Client functionality

Creating a port where Lisp is the DDE client

Create a `client-port` instance by calling **`make-instance`** on the `client-port` class. Available initargs are:

`:name`

an arbitrary object (typically a symbol) to name this port. Defaults to `:client-`*n*, where *n* is an integer that is auto-generated to make the name unique.

`:application`

the "service" name of the DDE server application that this port will connect to. This is often the filename of the executable image that is run, such as "progman" for the Program manager, but can be any arbitrary string that that server chooses to use. This value can be either the official DDE string or the keyword symbol whose print name is that string.

`:topic`

the particular topic of the application that this port will address. (To address multiple topics of the same application, you must create multiple port instances.) This value can be either a string or a keyword symbol.

These attributes can be changed after the instance is created using **`setf`** and the accessors **`port-name`**, **`port-application`**, **`port-topic`** (all are **`setf`**'able). The new values will be used if the port is closed and re-opened.

Example:

```
;; Create a port with the program manager acting as the DDE
;; server
(setq port1 (make-instance 'client-port
              :application :progman
              :topic :progman))
```

Functionality

`open-port` [Function]

Arguments: *client-port*

Package: dde

■ Activates this port. The port can later be closed, its application and/or topic optionally changed, and opened once again. Example:

```
(open-port port1)
```

`close-port` [Function]

Arguments: *client-port*

Package: dde

■ Deactivates this port. If this function is not used, all client ports are closed anyway when Lisp exits. Example:

```
(close-port port1)
```

`port-open-p` [Function]

Arguments: *client-port*

Package: dde

■ Returns non-`nil` if and only if the port is open.

`send-command` [Function]

Arguments: *client-port command-string* &key (*timeout* 1000))

Package: dde

■ Sends a DDE "Execute" message to this port's server application. Command-string should be whatever arbitrary string is expected by that DDE server. Many servers accept strings as they would be written in that application's macro language, sometimes surrounded by required square brackets.

■ Returns non-`nil` if the command is accepted, and `nil` if it is rejected. The optional timeout argument is in milliseconds. If the command is neither accepted nor rejected within this timeout period, then `nil` is returned.

■ Example:

```
;; Tell program manager to add a program icon f
;; or notepad.exe
(send-command port1 "[AddItem(notepad.exe,Test Item)]")
```

send-request [Function]

Arguments: *client-port item* &key (*link* :cold) (*timeout* 1000))

Package: dde

■ Sends a DDE "Request" or "Advice" message to retrieve information from a DDE server for a particular item.

■ If *link* is :cold (which is the default), then the return value is a list of strings derived from the single string returned by the DDE server (Lisp separates it into multiple strings where TAB characters or RETURN/LINEFEED pairs appeared in the server's string). If *link* is :hot or :warm, then the answer is not returned from this function call, but rather the **receive-advice** generic function will be called whenever the value for the requested item changes, assuming that the server responds to the changes as it should. If *link* is :stop, then a hot or warm link that was previously started will be stopped.

If the request is unsuccessful, as when the server doesn't handle the requested item, then nil is returned. The optional timeout argument is in milliseconds. If the request is neither accepted nor rejected within this timeout period, then nil is returned.

Many applications that act as DDE servers implement a topic called SYSTEM and an item called SYSITEMS, and sometimes an item called TOPICS, to which you can send a request to find out about additional topics.

■ Example:

```
;; Ask program manager for the names of all of
;; its group windows
(send-request port1 :groups)
```

receive-advice [Generic function]

Arguments: (*port* port) *topic item string*

Package: dde

■ If **send-request** was called with a link type of :hot or :warm, then this generic function will be invoked whenever the server tells us that the value for the requested item has changed. You may write methods on this generic function to handle these change notifications however it is appropriate for your application.

If the **send-request** link argument was :hot, then the *string* argument here will be a string; if the link argument was :warm, then the string argument will be nil.

active-client-ports [Variable]

Package: dde

■ A list of the ports that we have opened with any DDE servers.

send-value [Function]

Arguments: *client-port item value-string*
&key (*timeout 1000*)

Package: dde

■ Sends a DDE "Poke" message to send unsolicited information to a DDE server. Non-nil is returned if the server accepted the poke, and nil otherwise.

4.2 Server functionality

service-name [Variable]

Package: dde

■ The default service-name that will be established for the Lisp DDE server when **open-server** is called. If a different service-name is passed to **open-server**, then this variable will be set to that value. Initial value is :allegro.

service-topics [Variable]

Package: dde

■ The allowed topics with which clients can connect to the Lisp DDE server. The initial value is (nil :system :eval). An empty list indicates that any topic is allowed. A nil within the list indicates that the null topic is allowed.

This list will be returned if a client sends the Lisp server a request with the :system topic and the :topics item, to follow the convention for telling a client what topics are available on this DDE server.

If the :topics argument is passed to open-server, then this variable will be set to the value of that argument.

sysitems [Variable]

Package: dde

■ A list that will be returned when a client sends a request to our server with the :system topic and the :sysitems item, to follow the convention for telling a client what items are available on the :system topic. The initial value is

```
(:topics :command-result)
```

server-active-p [Variable]

Package: dde

■ This variable indicates whether our Lisp DDE server is currently active, as it is when **open-server** has been called and neither **close-server** nor **close-dde** has been called since then.

open-server [Function]

Arguments: &key (*name* *service-name*)
(*topics* *service-topics*)

Package: dde

■ Establishes this Lisp process as a DDE server. Any client can thereafter connect to this Lisp by using the service name and allowed topics that are established here. *name* and *topics* default to the global variables indicated. If other values are passed in, the global variables are set to the values that were passed.

■ Each Lisp process may open only one dde server, though multiple clients may each open multiple ports to it.

`close-server` [Function]

Arguments:

Package: `dde`

■ Makes this Lisp process no longer act as a DDE server. The server can be re-opened later after closing it.

`*active-server-ports*` [Variable]

Package: `dde`

■ A list of all ports that client applications have opened with this Lisp server. These server-port instances are created and managed automatically, but the list is available if you want to see how many applications are currently using this Lisp as a server. Unfortunately we cannot tell who those applications are.

`execute-command` [Generic function]

Arguments: *`topic command-string`*

Package: `dde`

■ This generic function is invoked when an application sends the Lisp server a command to execute. You may write methods on this generic function to execute arbitrary commands as appropriate. This method's action can depend on the particular topic name, which is always a keyword symbol. The value returned by this generic function can be retrieved later by the application if it sends a request with the `:command-result` item for this topic.

■ The following built-in method is defined:

```
(defmethod execute-command ((topic (eql :eval)) command-string)
   (let ((*read-tolerant* t))
      (eval (read-from-string command-string))))
```

For the special topic `:eval`, this built-in method executes the command string as a Lisp form. Note that this won't work in a runtime Lisp since it calls **`eval`**, which invokes the compiler. Your own **`execute-command`** methods for a runtime application should do some sort of command execution that you implement.

`answer-request` [Generic function]

Arguments: *topic item command-result*

Package: dde

■ This generic function is invoked when another application sends a request to the Lisp DDE server. You may write methods on this generic function to return a string as appropriate for the passed-in topic and item, which will always be keyword symbols. The default method returns the null string.

The *command-result* argument can normally be ignored -- it contains the result of the previous command that was executed for this topic by the application that is sending this request, and is passed here for the special version of this generic function that returns that value when the item is `:command-result`.

■ Here are some answer-request methods that are built into the allegro DDE facility, to respond to common DDE topics and items.

```
(defmethod answer-request ((topic (eql :system))
                           (item (eql :sysitems))
                           command-result)
   (list-to-tabbed-string *sysitems*))
```

This built-in **`answer-request`** method follows the convention for telling a client what items are available under the `:system` topic.

```
(defmethod answer-request ((topic (eql :system))
                           (item (eql :topics))
                           command-result)
   (list-to-tabbed-string *service-topics*))
```

This built-in answer-request method follows the convention for telling a client what topics are available on this DDE server.

```
(defmethod answer-request ((topic (eql :system))(item (eql :help))
                           command-result)
   "Send a DDE request for the HELP item of other topics for
    info on those topics")
```

This help item informs a client that it can get help about topics other than the system topic by sending a request with the HELP item to those topics.

```
(defmethod answer-request ((topic (eql :eval))(item (eql :help))
                           command-result)
   #.(format nil "Send a DDE Execute using this EVAL topic ~
      to evaluate an arbitrary lisp form.  To get the value ~
      returned by that form, send a DDE Request using this EVAL ~
      topic with the item COMMAND-RESULT.  To retrieve the value ~
      of any lisp symbol, send it as ~
      the item in a DDE Request using this EVAL topic."))
```

This help item for the eval topic explains how to use the eval topic.

```
(defmethod answer-request ((topic (eql :eval)) item
                           command-result)
   (format nil "~s" (symbol-value
                     (intern (symbol-name item) *package*))))
```

For the special topic `:eval`, return the value of the symbol named by the item argument in the current package

```
(defmethod answer-request ((topic (eql :eval))
                           (item (eql :command-result))
                           command-result)
   (format nil "~s" command-result))
```

For the special item `:command-result` of the `:eval` topic, return the value that was returned by the most recent **`execute-command`** method that was invoked for this client application

`post-advice` [Function]

Arguments: *topic item*

Package: dde

■ A DDE server Lisp should call this function whenever an item for which that Lisp server can handle hot or warm links has changed. This will result in the server Lisp's answer-request generic function being invoked for any items for which hot or warm links are currently established.

receive-value [Generic function]

Arguments: *topic item value-string*

Package: dde

■ This generic function is invoked when an application "pokes" a value to the Lisp DDE server. You may write methods on this generic function to process the poked values as needed. If you consider the value to be "accepted" by your Lisp application, then you should write the method to return non-`nil`, or else `nil` to tell the client that you have rejected the poked information.

The default **receive-value** method for all topics interprets the *item* argument as a Lisp symbol in the current package, and sets its value to be a Lisp object that is read from the *value-string* passed in.

list-to-tabbed-string [Function]

Arguments: *list*

Package: dde

■ A utility function that prints a list of Lisp objects to a string, with tab characters between successive objects, to facilitate the DDE convention of returning multiple answers as a single tab-delimited string. A handy function to use inside **answer-request** methods.

Chapter 5 Windows typedefs and API's

Many C types are already defined in Allegro CL. They are defined with **`defctype`**, **`defcstruct`**, etc. forms. The arguments and return types used by predefined Windows API functions (listed after the types) use the types here defined. The file *osidoc.txt*, in the distribution directory included, contains these forms as well as the list of pre-defined Windows API functions. To find specific definitions, bring that file up in a editor window and use editing tools to search through it.

In this chapter, we show small pieces from the file to show what it looks like.

Typedefs

These forms defines types available in Allegro CL for Windows. Primitives are defined first, with forms such as these:

```
(defctype int :long)
(defctype uint :unsigned-long)
(defctype bool :long-bool)
(defctype byte :unsigned-char)
(defctype char :char)
(defctype ushort :unsigned-short)
(defctype word :unsigned-short)
...
```

Following the simple definitions are more complex forms based on the primitive definitions. Here are some examples.

```
(defcstruct box
  ((left            long)
   (top             long)
   (right           long)
   (bottom          long)
  ))

(defcstruct position
  ((x               long)
   (y               long)
  ))

(defctype void :void)

(defcstruct stat
  ((st_dev          short)
   (st_ino          ushort)
   (st_mode         ushort)
   (st_nlink        short)
   (st_uid          short)
   (st_gid          short)
   (st_rdev         short)
   (st_size         long)
   (st_atime        long)
   (st_mtime        long)
   (st_ctime        long)
  ))
```

Allegro CL access to Win32 API

Many functions in the Windows Application Programmers Interface (the Win API) are already defined within Allegro CL for Windows. These pre-defined functions are listed in the file *osidoc.txt* in the distribution directory. To examine the list, open that file in a text-edit window.

We reproduce a small part of the file here. The first column is the return value. The second column is the function name and argument list. The types are in the type definition portion of the file *osidoc.txt*. The list in the file is arranged alphabetically by the function name (that is, the second column is alphabetical).

All the symbols naming the functions are in the `win` package.

Here is a simple example, suppose you wanted to know the maximum allowable time (in milliseconds) that may occur between the first and second click of a double click. You know from your Win API book (e.g. the MS Windows Software Development Kit *Programmer's Reference*, volume 2 *Functions*) that the API **`getdoubleclicktime()`** returns that value. In Lisp, you would call:

```
(win:getdoubleclicktime)
```

Looking in the table, you see this function returns a `uint` (the Lisp equivalent of an unsigned integer). On our system, this function returned 452.

Note that some lines wrapped in this table, since the functions took a lot of arguments. There is no other significance to the line wrapping.

```
int             abortdoc(hdc);
(char *)        aclpc_kernel_version((char *),short);
atom            addatom(lpctstr);
int             addfontresource(lpctstr);
bool            adjustwindowrect(lprect,dword,bool);
bool            adjustwindowrectex(lprect,dword,bool,dword);
bool            animatepalette(hpalette,uint,uint,(paletteentry
                *));
bool            anypopup();
bool            appendmenu(hmenu,uint,uint,lpctstr);
bool            arc(hdc,int,int,int,int,int,int,int,int);
uint            arrangeiconicwindows(hwnd);
handle          begindeferwindowpos(int);
hdc             beginpaint(hwnd,lppaintstruct);
bool            bitblt(hdc,int,int,int,int,hdc,int,int,dword);
bool            bringwindowtotop(hwnd);
bool            buildcommdcb(lpstr,lpdcb);
bool            callmsgfilter(lpmsg,int);
lresult         callwindowproc(wndproc,hwnd,uint,wparam,lparam);
bool            changeclipboardchain(hwnd,hwnd);
short           chdir((char *));
bool            checkdlgbutton(hwnd,int,uint);
dword           checkmenuitem(hmenu,uint,uint);
bool            checkradiobutton(hwnd,int,int,int);
```

```
hwnd            childwindowfrompoint(hwnd,point);
short           chmod((char *),short);
bool            choosefont(lpchoosefont);
bool            chord(hdc,int,int,int,int,int,int,int,int);
int             chsize(hfile,long);
bool            clearcommbreak(handle);
bool            clienttoscreen(hwnd,lppoint);
bool            clipcursor(lprect);
bool            closeclipboard();
handle          closemetafile(hdc);
bool            closewindow(hwnd);
int             combinergn(hrgn,hrgn,hrgn,int);
long            commdlgextendederror();
handle          copymetafile(handle,lptstr);
int             copyrect(lprect,lprect);
int             countclipboardformats();
hbitmap         createbitmap(int,int,uint,uint,lpstr);
hbitmap         createbitmapindirect(lpbitmap);
hbrush          createbrushindirect(lplogbrush);
bool            createcaret(hwnd,hbitmap,int,int);
hbitmap         createcompatiblebitmap(hdc,int,int);
hdc             createcompatibledc(hdc);
hcursor         createcursor(handle,int,int,int,int,lpstr,lpstr);
hdc             createdc(lpstr,lpcstr,lpcstr,lpdevmode);
hwnd            createdialog(handle,lptstr,hwnd,dlgproc);
hwnd            createdialogindirect(handle,lpstr,hwnd,dlgproc);
hwnd            createdialogindirectparam(handle,lpstr,hwnd,
                dlgproc,lparam);
hwnd            createdialogparam(handle,lpctstr,hwnd,dlgproc,
                lparam);
hbitmap         createdibitmap(hdc,lpbitmapinfoheader,dword,lpstr,
                lpbitmapinfo,uint);
...
```

Index

Numerics

A

C

D

E

F

G

H

L

N

O

P

R

S

U

V

W

PREFACE

There are two volumes of bound documentation for Allegro CL for Windows. This is volume 2. Each volume contains several manuals. There is also a *Read This First* document, which not bound in with the rest of the documentation.

Here is a brief description of the documentation:

1. *Read This First*. This document is supplied loose. It contains information that was not put in the bound documentation.

Volume 1

2. *Getting Started*. This document describes how to install Allegro CL for Windows on your system and it gives information about running the product.
3. *Common Lisp Introduction*. This document is an introduction to the Common Lisp language and the Allegro CL for Windows implementation of it.
4. *Interface Builder*. The Interface Builder allows you to build an interface to your application interactively. It is described in this manual.
5. *Foreign Function Interface*. Allegro CL for Windows supports calling applications written in languages other than Lisp. This document describes the interface.

Volume 2

6. *Programming Tools*. This document describes the user interface to Allegro CL for Windows. In particular, it describes the Toploop window, the editor, the debugging facilities, etc.
7. *General Index*. An index to all documents in Volumes 1 and 2.

Professional version only

The Professional version of Allegro CL provides the facility to create standalone applications. User who purchase the Professional version also receive the following document:

Professional supplement. This document describes features available in the Professional version (but not in the standard version), including how to create standalone applications.

Each individual manual has a table of contents at the beginning and an index at the end. The *General Index* manual, in volume 3, is an index for all manuals in Volumes 1 and 2.

Allegro CL for Windows

Programming Tools

version 3.0

October, 1995

Copyright and other notices:

This is revision 0 (initial version) of this manual. This manual has Franz Inc. document number D-U-00-PC0-05-51010-3-0.

Copyright © 1992, 1995 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademark of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

Contents

[This page intentionally left blank.]

PREFACE

This manual *Programming Tools* describes the user interface to Allegro CL for Windows. We recommend that every user of Allegro CL at least scan this manual and keep it handy while using Allegro CL.

Chapter 1 in particular describes starting Allegro CL, initialization (startup) files, and getting online help. Chapter 2 describes the Toploop window (used for typing to Lisp). Note section 2.14 which describes setting your preferred Lisp environment.

Chapter 3 is also important for all users. It describes the text editor. The remaining chapters describe additional tools.

Sections at the end of most chapters describe the internals of programming tool described in the chapter. This information was contained in the manual *Inside Programming Tools* in earlier releases, but that manual has been folded into this one in release 3.0. You can use the information in those sections to customize or modify the tools described. New users, however, should note that customization and modification are not required.

There are online code examples supplied with the Allegro CL distribution. Many that relate to programming tools are in the *ex\ext* directory read off the distribution disks (typically in the *allegro* directory if you followed the default installation instructions). You can try out those examples as you read the text.

[This page intentionally left blank.]

Chapter 1 Introduction

Organization of this Guide

This Guide tells you how to use the Allegro CL for Windows programming tools on the Microsoft Windows Operating System on PC's. The tools enable you to create, edit and debug your Lisp programs quickly and efficiently in the Lisp environment. This Guide contains the following chapters:

1. **Introduction**, the chapter you are now reading. The remainder of this chapter describes the terminology and conventions used in the Guide. It also explains how to use the online help facilities while you are programming.
2. **The Toploop**. This chapter explains how to use the Toploop, the top-level read-eval-print loop through which you interact with the Lisp system.
3. **The Text Editor** explains how to use the Text Editor to create and edit programs, and how to save your programs as documents on disk.
4. **The Inspector** explains how to use the Inspector to examine and modify Lisp data structures.
5. **Trace, Breakpoint and Profile** explains how to use the Trace and Breakpoint facilities of Allegro CL for Windows. You can trace your programs and set breakpoints to help you track down bugs. The Profile facility is also described: this allows you to measure the performance of your programs.
6. **The Debugger** explains how to use the Debugger, which allows you to examine and modify the Lisp stack at a certain point in the execution of your program (for example when an error occurs).
7. **The Stepper and the Watch Facility** explains how to use the Stepper to single step through your programs, and how to use the Watch facility to monitor places as they are updated.

8. **The CLOS browser**. There are several graphical tools available for assisting with CLOS. They are described in this section.
9. **Comtabs**. This chapter describes the command tables (comtabs) which form the basis for adding commands to the text editor.

There is an appendix that gives key mappings for the various editor modes. Finally, there is an index.

Throughout this Guide, it is assumed that you are familiar with the basic operation of the PC and of the particular versions of the Windows Operating System that you are using. It is also assumed that you have installed Allegro CL for Windows on your computer. For details of how to do this, see the *Getting Started* manual in volume 1 of Allegro CL for Windows printed manuals.

1.1 Notation

We use fonts, display format, and special characters to convey information in this and other Allegro CL for Windows manuals. However, we have tried to keep use of different fonts and special characters to a minimum.

Fonts

Most of the text in this manual is 11 point Times Roman. We use the following special fonts for the following purposes:

Font	Purpose
`courier bold`	operators (functions, macros, special forms). This font is used exclusively for symbols naming operators.
`courier regular`	variables, symbols (except when naming an operator), code samples and fragments.

Font	Purpose
courier slant	Arguments and placeholders. This font is used when you (the user) will insert something different from what appears in the text.
Italic	Filenames, names of books and manuals, and emphasis. This font is also used to identify a word being defined (e.g. 'A *symbol* in lisp is ....')
Bold	Section and chapter titles. This font is also used for emphasis, typically when a whole sentence or paragraph rather than a single word is emphasized.

Some notes on font usage:

- The same symbol can be used both to name an operator (function, macro, etc.) and to name another Lisp object (a type, a variable, etc.) We use **courier bold** when and only when the symbol is being used to name an operator.
- We do not use courier bold in code samples and fragments since in such cases, operators can be identified syntactically. Thus in the form `(car '(apply cdr))`, `car` must name a function and `cdr` and `apply` are elements of a list.
- We do not typically use special fonts for types except when following the word type. Thus, we say 'the argument must be a fixnum' and 'the argument must be of type `fixnum`'.

Change bars

You will see change bars (a black line to the left of the text) from time to time in this manual. They indicate that the text has been changed in some significant way (from the 2.0 version of this manual). Text which is simply corrected and or rewritten but the information is the same does not have change bars.

1.2 Getting help

There are numerous facilities for getting help while using Allegro Common Lisp. We discuss a number of them in this section.

The context-sensitive right-button-menu

Depressing the right mouse button while the cursor is over some object displays a menu of choices appropriate to the object. The picture below shows the right-button menu displayed while the cursor is over `setq` (after `(setq` has been typed to the toploop window).

The first item in the menu calls for an inspection of the named object (the symbol `setq`). The remaining items are applicable to a symbol (**Apropos**, **Describe**, **Find-in-File** -- which looks for the symbol definition in a source file) while others are applicable to the window itself (**Print** -- prints the contents of the window, **Find-Replace** -- which displays a submenu of various find/replace options). Over other objects, the right button menu is quite different. When the cursor is over the Status Bar, the menu contains options to hide the Status Bar, or change its font or size.

Finding a symbol given only part of its name

The **Apropos** choice from the right-button menu (or from the Help menu) causes the output of the form (apropos <symbol>) to be printed to the Help window (creating it if necessary). The Lisp function **apropos** prints all symbols whose print-names contain the print-name of the selected symbol. Choosing **Apropos** while the cursor is over `setq`, for example, results in the following in the Help window:

The format of the display in the window is:

`SYMBOL-NAME ; status-as-variable, status-as-operator`

> *status-as-variable* is either unbound or bound as the named symbol does or does not have a value.
>
> *status-as-operator* shows whether the function slot of the symbol defines the symbol as naming a function, a macro, a special form, etc. If no operator is named by the symbol, *status-as-operator* is blank.

In the illustration, all symbols shown are unbound and variously name functions, macros, and special forms, and some have no function binding. Look for **apropos** in the Online Manual for more information.

The Help window

All the help commands display their results in the Help window. The window will retain the contents of previous help requests. Like any window, you can bring it to the top by choosing **Help** from the Windows menu. The Help window is a Text Edit window so you can print a copy of the Help window or save it like any other Text Editor window.

Other right-button and Help menu choices over symbols

When the cursor is over a symbol, other options on the Help menu and/or the right-button menu include:

Describe: prints the output of (describe <symbol>) to the Help window. See **`describe`** in the Online Manual.

Documentation (Help menu only): prints the documentation string (if there is one -- most Allegro CL for Windows symbols do not have a documentation string) to the Help window (see **`documentation`** in the Online Manual).

Lambda-List: prints the lambda-list of the operator named by the symbol to the Help window.

Quick-Symbol-Info: prints a brief description of the symbol to the Status Bar.

Quick Class Info (not applicable to setq so does not appear in right-button menu): prints a brief description of the class named by the symbol to the Help window.

Manual Entry (Help menu) **Windows-Help** (right-button menu): brings up the entry on the symbol in the Online Manual.

Manual Contents (Help menu): displays the contents of the Online Manual.

Return-Selected-Object (right-button menu and Tools/Miscellaneous menu): returns the selected object (in this case, the symbol) in the Toploop window. While this choice is not that useful for symbols, it is useful when another Lisp object is the selected object since after being returned in the Toploop window, the object is the value of the variable `*`.

About Allegro CL (Help menu only): displays a small window containing information about Allegro CL.

Information in the Status Bar

The Status Bar is the pane at the bottom of the screen when Allegro CL for Windows comes up. It can be displayed or hidden with the F11 key, or from the Toolbar/Status Bar submenu of the Tools menu. When it is visible, information is constantly displayed in the Status Bar about whatever is under the cursor. Information is displayed about every button on the Toolbar, about most buttons on Allegro CL for Windows dialogs, about the symbol nearest the cursor (or the selected object) in a text-edit window (such as the Toploop window), etc. We advise people to keep the Status Bar visible.

Lambda-lists in the Status Bar

When you start typing a form in a text-edit window (such as the Toploop window), the lambda-list (argument list) of the operator being called by the form is displayed in the Status Bar as soon as the space after the operator name is entered. Thus, typing '`(setq ` causes

Common-Lisp Special Form SETQ var value &rest other-vars+values

to be displayed in the Status bar, telling you that **`setq`** is a special form and its lambda-list is 'var value &rest other-vars+values' (i.e. a variable name, a value, and optionally other variable name/value pairs). If you continue typing, say entering `*print-level*` and a space, information on `*print-level*` is displayed (that it is a Common Lisp variable and its current value).

If you want to see the lambda-list for **`setq`** again, enter an *extra* space and it reappears.

Online manual

The **Manual Entry** and **Manual Contents** items bring up the Online Manual. This describes the Lisp language including CLOS and Common Graphics. Manual Entry (in the Help menu, Windows-Help in the right-button menu) brings up the entry for the symbol nearest the text cursor in the active window (if there is no such symbol or if the nearest symbol does not have an entry, that fact is printed to the status bar and the Online Manual does not come up). Manual Contents brings up the contents page of the Online Manual.

Other items on the Help menu

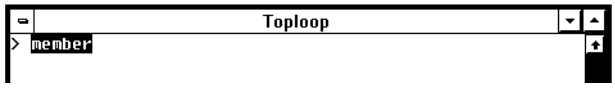
Quick Symbol Info displays information about the symbol (including the lambda list if the symbol names a function or other operator) in the status bar.

Technical note on the Help window

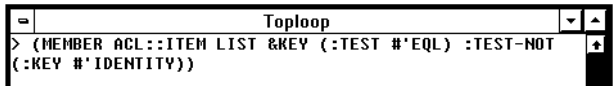
You can use the help commands from within your programs. The Help window is available as the stream `allegro:*help-output*`.

Templates For Building Function Calls

The **Build Call** on the Tools/Miscellaneous menu command provides a handy way of constructing a template for calling a function. It may be used to replace a selected symbol in an editor window with a list consisting of the symbol followed by its lambda list. As an example, type `member` into the Toploop window and select it (we have cleared earlier work from Toploop):



Choose **Build Call** from the Miscellaneous submenu of the Tools menu. This puts the template for member onto the Clipboard window. Now choose **Paste** from the Edit menu. Lisp replaces the selected symbol with a template of a call to it:



You can now edit the template into an actual call.

Garbage Collection

Whenever a garbage collection occurs, the cursor changes to a small hourglass. The hourglass remains until the garbage collection is completed, then the cursor returns to normal.

Chapter 2 The Toploop

2.1 Starting Lisp

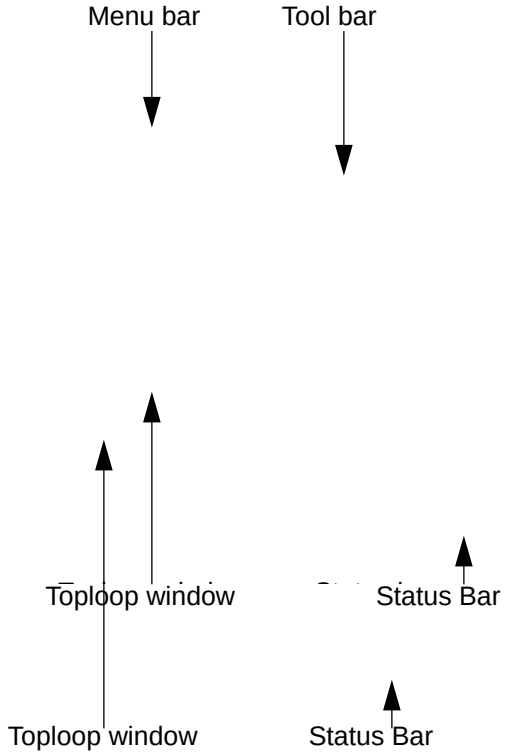
Allegro CL for Windows must be installed on a hard disk to run. The technique for starting programs (of any type) depends on the type of Windows you are running (3.1, NT, 95) and on your mode of running it. Thus, if you use the Windows 95 shell (standard with Windows 95 and possible with Windows NT)), click on the Start button and choose Programs/Allegro CL 3.0 for Windows/Allegro CL from the submenus that appear. If you use a Program manager (standard with Windows 3.1), double click on the application icon to start Lisp. If you use a different shell or method, start Allegro CL as you do other programs.

It is possible to save an image (by choosing **Save Image...** from the File menu -- see section 2.15 **Images** below). To start the saved image, you first start Allegro CL in the usual way and then choose **Load Image...** from the file menu. Specify the saved image file (which will have extension *.img*) to the dialog that appears.

2.2 Tools available on startup

After Allegro CL for Windows starts up, its application windows should look something like the following picture. But not exactly like. You will see colors if your machine has a color monitor. The [date and time] in the Toploop will be a real date and time (they are the

date and time when the distribution was created and serve to exactly identify the distribution you received). Information about patches will appear in the banner (the text printed before the initial prompt) if patches are loaded. And there may be other minor differences.)



The four labeled features are described in sections 2.3 **The Toploop window**, 2.5 **The menu bar**, 2.6 **The toolbar**, and 2.7 **The status bar**.

2.3 The Toploop window

The Toploop itself is not so much a tool as an integral part of the Lisp system. Most interaction with Lisp proceeds via the Toploop, although not necessarily through the Toploop window. The Toploop is so called because it is the highest level of control in the Lisp ses-

sion and consists in essence of a loop which reads expressions from windows, evaluates them and then prints the results. It also manages other facilities such as a history list and Clipboard.

You can cause a new Toploop to begin executing by calling the **`toploop`** function. For details of how to do this and other aspects of controlling the Toploop, refer to section 2.16.5 below in this chapter.

The title of the window appears in the bar at the top. Throughout this Guide, phrases such as “the Toploop window” are used to abbreviate “the window whose title is Toploop” and also “the window through which interaction with the Toploop occurs”.

Initial package

When Lisp starts up, the current package is the `common-lisp-user` package (nicknamed `cl-user` and `user`). This package uses the `common-lisp`, `allegro`, and `common-graphics` packages.

Startup files

When Lisp is started up it looks for two files in the directory where the image is located and loads them if it finds them. The two files are:

- *prefs.lsp* or *prefs.fsl* (i.e. a Lisp source file or a compiled Lisp file with name *prefs*). This file must be written by choosing **Save** from the Preferences dialog (displayed by choosing **Main Preferences** from the Preferences menu).
- *startup.lsp* or *startup.fsl* (i.e. a Lisp source file or a compiled Lisp file with name *startup*). You create this file and you can put any valid Lisp forms in it.

The *prefs* file is loaded first. The forms in this file are read and evaluated using the **`load`** function. The *prefs* file is written by Allegro CL for Windows based on how you customize your environment while Lisp is running. You typically start Allegro CL for Windows the first time, set things up to your liking and then bring up the Preferences dialog (by choosing **Main Preferences** from the Preferences menu). The dialog (which is a tab widget with 8 tabs) reflects the current state of your Lisp and has buttons to save *prefs.lsp* (and to apply changes made in the dialog to Lisp when you have changed values).

You can create a *startup* file by choosing **New** from the Files menu (which creates a Text Edit window named Untitled), putting the forms you want in it, and choosing **Save** from the File menu. When you are prompted in a dialog box for a filename to save to, enter *startup.lsp*.

You can compile either file by choosing **Compile...** from the file menu after you have written the sources.

Please note the following:

- Because the files are loaded by **`load`**, you cannot change the current package from within the file (since **`load`** binds `*package*`).
- You do not need to have either file. If Lisp is unable to find either file or both files, it starts up without them.
- It is common to put a form similar to `(format t "Read startup~%")` in the startup file to remind yourself that the file was read. (The message will be printed in the Toploop window when Lisp starts up.)
- **Large startup files**. Loading a large startup file can take a long time. You may prefer to save a new image and start Lisp from that, instead of loading Lisp Startup every time.
- **Technical note**. The `*session-init-fns*` may be used to achieve a similar effect to Lisp Startup. See the description of that variable in the Online Manual.

Typing

The Toploop window initially contains the banner, a greater-than sign (>) and a blinking vertical bar. The greater-than sign is a prompt displayed by the Toploop to indicate that it is ready to take part in an interaction. Lisp displays a new prompt at the end of the Toploop window after the completion of every interaction.

The blinking vertical bar is the insertion point into the window. You can move the insertion point using the arrow keys or by moving the pointer and clicking. For ease of accurate positioning, the pointer is shaped like an I-beam when it is inside the window. When you type on the keyboard the characters will be inserted at the insertion point. The Backspace key deletes the character to the left of the insertion point while the Delete key deletes the character to the right of the insertion point. Pressing the Enter key (next to the alphabetic keyboard) causes the insertion point to be moved down to the next line and a suitable indentation to be provided, to help make code typed into the window more readable. If a complete form has been entered, pressing Enter will cause it to be evaluated. Pressing Alt-Enter (equivalently, the Enter key next the numeric keypad) has the same effect as Enter when typing to the Toploop window.

At any time, you can erase anything in the Toploop window by either positioning the text pointer just beyond the character(s) you want to delete and hitting the backspace key as many times as required. Or you may simply select what you want to erase and then press either the Backspace or the Delete key. Besides deleting selected text, the Delete key deletes character in front (rather than behind) the cursor.

Certain key combinations cause special actions to take place, such as performing Editor operations or executing menu commands. These generally involve the Alt key being depressed in conjunction with other keys and will not be discussed in detail here.

> **Text Editor commands**. Because the Toploop window is a Text Editor window, any Text Editor command can be used in the Toploop window. The Text Editor is described in Chapter 3 of this Guide.

The copy down feature

If you place the cursor at the beginning of a previously evaluated form and press Enter, the form is copied down to the latest toploop prompt. Pressing Enter again evaluates the form. You can edit the form as well, and evaluate it by positioning the cursor at the end and pressing enter.

Finding the Toploop prompt

It is possible to delete the prompt in the Toploop window (typically by mistake). It also may happen that you cannot (easily) find the Toploop window or the end of the Toploop window. In any case, evaluating (anywhere)

```
(top:find-toploop-prompt)
```

causes the Toploop window to be exposed, the cursor to move to the bottom of the window, and a new Lisp prompt to be generated. Further, each built-in comtab binds Control-Shift-N to this function, so that key combination will expose the Toploop window and produce a new prompt whenever the input focus is in any text window (such as the Toploop window, file editors, or the Help window). Choosing **New Toploop Prompt** from the Tools menu also calls this function, as does clicking over the leftmost (in the default) toolbar button (the one with the > sign).

Symbol completion

When typing in the Toploop or other Text Edit window, pressing Control-. will cause a window of possible completions to be displayed. For example, if you have typed:

```
(memb
```

Pressing Control-. causes a window to be displayed with the following items:

a Member

b Member-If

c Member-If-Not

e Memberq

f Memberql

Each symbol that the system knows about that is a potential completion is displayed in this window. Pressing the letter associated with a choice or double-clicking over a choice causes the symbol to be completed in the Text Edit window. Thus, if we enter **b**, the completion `member-if` is entered and we can continue.

The search will start with symbols available in the current package. If no completion is available, other packages will be searched if the variable `acl:*symbol-completion-searches-all-packages*` (documented in the Online Manual) is

true. (The correct package qualifier will be added automatically.) If that variable is `nil`, the search will stop without considering other packages. Note that the search of other packages can take quite a while. Pressing Escape will abort the search.

If there is only one completion, it is entered without a window popping up. If there are more than `*max-symbol-completion-choices*` (initial value 50, documented in the Online Manual) possibilities, that fact is reported in the status bar and no window is displayed.

You can reject all suggested completions by clicking away from the Completions window. If there are no completions, that fact is reported in the Status Bar and no window is displayed.

Using super-brackets

Normally you write Lisp expressions using parentheses, (and). Allegro CL for Windows also has super-brackets represented by the square brackets [and]. A closing super-bracket closes off all opening parentheses until either there are none left or an opening super-bracket is found. Thus the following expressions are all equivalent:

```
(defun foo (a b) (cons (list a) b))
(defun foo [a b] (cons (list a) b)]
(defun foo (a b) (cons (list a) b]
```

An opening super-bracket must be matched by a closing super-bracket, not a closing parenthesis. Closing super-brackets are very convenient at the end of a function definition when you simply want to close all open parentheses. Used sparingly, they help you to avoid unbalanced parentheses errors in the Text Editor and make your programs easier to type in, but used to excess they can be confusing and a source of errors. You should regard them as a useful shorthand notation.

Note that super-brackets are not standard Common Lisp so code that uses them will not be portable to other implementations.

2.4 Exiting From Lisp

The **Exit** command on the File menu is used to exit completely from Lisp:

If you attempt to exit when the variable `*top-query-exit*` is non-`nil`, Lisp asks you if you really want to exit, and gives you the option of not doing so. This acts as a safety feature in case you inadvertently choose **Exit**. You may change the value of this variable either by using **`setq`** in the usual way, or by using the Toploop Preferences dialog as described in section 2.14.

Another way to exit from Lisp is to close all Toploop windows, as described under the next heading.

Closing the Toploop window

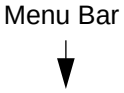
The **Close** command on the File menu is described with the Text Editor in Chapter 3, but it behaves slightly differently when Toploop is the active window. Choosing **Close** causes the most recent version of the Toploop to be exited. There may be several Toploops active at

once, created for instance by using the **`toploop`** function in a Lisp program (see section 2.16.5 below). Attempting to exit from the outermost Toploop invokes the **Exit** checking mechanism to make sure that you really want to leave Lisp.

> **Shortcut**: you can close a window by double-clicking in the Control-menu box in the upper left corner.

2.5 The menu bar

The Allegro CL menu bar displays menus available while using Allegro CL for Windows. The menus are standard Windows menus.)



You can display a menu with the mouse (by clicking on the menu title in the menu bar). You can also display them from the keyboard by pressing the Alt key and the underlined letter in the menu title (F for the File menu, E for the edit menu, etc.) Keep the Alt key down while the menu is displayed. Most entries will have an underlined letter and pressing that letter while the Alt key is down is equivalent to choosing the menu item.

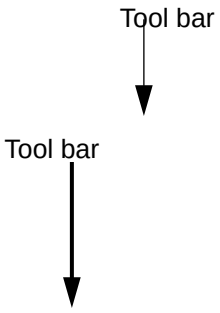
It is possible to add your own menus to the menu bar. See the description of menus in the *Common Graphics* section of the Online Manual for more information.

The contents of menus are fairly self-explanatory. If a menu item acts on a window or on a selected piece of text, it is the currently active window or currently selected text that is acted upon.

Note that many choices, particularly in the Help and Tools menus, act on the selected object or the object nearest the text cursor. Thus, choosing **Online Manual** from the help menu displays the online documentation for the symbol nearest the text cursor.

Keep you eye on the status bar, where messages about what Lisp is trying to do and, when it cannot do something, explanations of why it can’t are printed.

2.6 The toolbar



The toolbar is a line of button widgets displayed under the menu bar. Clicking a button causes a specific action, often one that can also be caused be choosing a menu item from some menu. For example, the leftmost button (displaying a >) exposes the toploop window and moves the cursor to a fresh prompt, just like choosing **New Toploop Prompt** from the Miscellaneous submenu of the Tools menu.

How to display or hide the toolbar

The toolbar is displayed automatically when Lisp starts up. Choosing **Toggle Toolbar** from the submenu displayed after choosing **Toolbar/Status Bar** from the Tools menu will hide the toolbar if it is displayed and display it if it is hidden. Pressing the F12 key has the same effect as choosing the menu item.

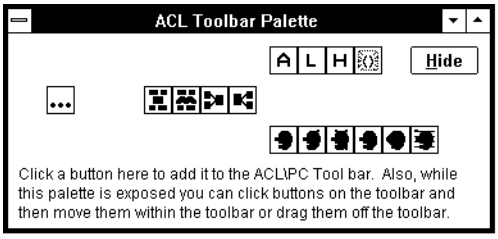
Note that the toolbar will be in front of any window whose parent is `*lisp-main-window*`. You may on occasion wish to hide the toolbar in order to see more of a window that is partially obscured by the toolbar. The F12 key is particularly useful for this purpose.

What the toolbar buttons do

As you move the mouse cursor over the buttons on the toolbar, look at the status bar, where the effect of clicking over the toolbar button is described. Also, if the Tooltips facility (see the description of Tooltips in the Online Manual) is on, a small tooltip pop-up window with a word or two describing the button is displayed when the cursor is over the button.

Editing the toolbar

Choosing **Toolbar Palette** from the submenu displayed by choosing **Toolbar/Status Bar** from the Tools menu displays the following dialog:

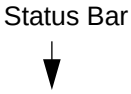


While this dialog is displayed, the toolbar is in edit mode, meaning clicking over the buttons do not have their usual effect. Instead, pressing the left mouse button over a toolbar button allows you to move it, either around the toolbar or to the toolbar palette. There are several extra buttons on the toolbar palette which you can add to the toolbar if you wish.

To see what the extra buttons do, move the mouse cursor over them and look in the status bar, where a description will be printed.

When you have finished editing, click on the **Hide** button. Once the palette is hidden, the toolbar returns to normal operation.

2.7 The status bar



The status bar is a static text dialog that appears at the bottom of the screen when Lisp is running. Messages printed there provide useful information while Lisp is running. For example, the following information is displayed:

- When the mouse cursor is over a button on the toolbar, the status bar contains a description of the effect of clicking over the button.
- When choosing a menu item has no effect, a reason will typically be displayed in the status bar. (For example, choosing Manual Entry from the Help menu should display the entry in the Online Manual for the symbol nearest the text cursor. if there is no such symbol, or if the nearest symbol is not described in the Online Manual, that information is displayed in the status bar.)
- When entering a form, typing a space after the operator causes the argument list for the operator to be printed in the status bar. The contents of the Status bar may change as you type in argument values but typing an extra space or Control-Z will cause the argument list to return.

Note that the message in the status bar often changes upon moving the mouse, typing a character, or choosing a menu item. Do not expect the message to stay in the status bar for long. If you need time to read the message, avoid moving the mouse or typing.

How to display or hide the status bar

The status bar is displayed automatically when Lisp starts up. Choosing **Toggle Status Bar** from the submenu displayed after choosing **Toolbar/Status Bar** from the Tools menu will hide the status bar if it is displayed and display it if it is hidden. Pressing the F11 key has the same effect as choosing the menu item.

Note that the status bar will be in front of any window whose parent is `*lisp-main-window*`. The F11 key is useful for temporarily hiding the status bar to expose something else.

Details of messages in the status bar

Two variables, `cg:*lisp-message-print-length*` and `cg:*lisp-message-print-level*` control the printing of lists in the status bar. They are formally defined in section 2.16.6 **The status bar** below. They work like `*print-length*` and `*print-level*`: when the value is a positive integer, only that many items (for length) or levels (for level) are printed in a list, with additional items or levels indicated by suspension points (...). If the value is `nil`, all items or all levels are printed.

Sending your own message to the status bar

The function **`cg:lisp-message`** will print a message in the status bar. Its argument list is (*`format-string`* `&rest` *`format-args`*). It is formally defined in section 2.16.6 **The status bar** below. It is not useful to call this function at the top level (since the status bar prints the result of a function evaluated at the top level, that message will immediately replace your message). Instead, you can use this function to print messages in the status bar when, say, the mouse passes over a dialog item in one of your dialogs (see the Common Graphics section of the Online Manual for more information).

Modifying the status bar

You can change the number of lines in the status bar or the font used for messages. Dialogs to accomplish these ends are displayed by choosing **Status Bar Lines** or **Status Bar Font** from the **Toolbar/Status Bar** submenu of the Tools menu or the associated choices from the right-button menu when the cursor is over the status bar.

2.8 Evaluating expressions

Pressing Alt-Enter when a complete form is selected in an editor window causes the Lisp reader to read an expression and print the value in the Toploop window. Note that the form itself is not complied to the Toploop window.

In the Toploop window, pressing Enter (or Alt-Enter) at the end of a complete form has the following effect:

- If the insertion point is in the Toploop window and after the most recent Toploop prompt, the reader will back up to the prompt and then read a single expression. Lisp will evaluate the expression and print its results at the end of the Toploop window followed by a new prompt.
- If the insertion point is in the Toploop window but before the most recent prompt, the reader will read a single expression beginning at the insertion point and then copy this expression to the end of the Toploop window, placing the insertion point after it. This action is termed *copy down* and is useful for maintaining a transcript of the session in the Toploop window and for repeated evaluations of similar forms (since the form can be edited after it is copied). Pressing Enter again causes Lisp to evaluate the expression.

Dialog boxes

If the active window is a dialog box (such as the Save dialog box illustrated below), it will often have a large button containing a word like **OK**. Pressing Enter is equivalent to clicking that button with the mouse. Note that if the dialog is modal (many pop-up dialogs are), you cannot type to another window until you have chosen **OK** (or pressed Enter) or **Cancel** (another typical button, meaning return to the state before the dialog was displayed). The following dialog, displayed when you select **Exit** from the File menu, is a modal dialog.

Interrupting Lisp

Pressing the Break key (sometimes labeled ‘‘Pause Break’’) will interrupt the evaluation of a form and give you the option of aborting the computation. You may have to hold the key down for a few seconds and it may take some more seconds for the interrupt to be processed. When that happens, a Restarts dialog box will appear announcing the interrupt. A Restarts dialog is illustrated below.

The `read` function

One situation in which the Toploop does not have control is when the reader is invoked by calling the **`read`** function (see the description on **`read`** in online help). The expression read is then returned by **`read`**.

Dealing with errors

If Lisp detects an error while reading or evaluating a form it will display a dialog box containing a description of the error, the available restarts, and several buttons. For example, evaluating the form `(car 8)` signals an error and displays the following dialog box:

In a Restarts dialog box, there are usually buttons labelled **Enter Debug** and **Abort** and. Clicking **Abort** will normally return you to the Toploop prompt. Clicking **Enter Debugger** will enter the Debugger. The available restarts are listed in a pane in the middle of the dialog. The restarts usually include equivalents of the **Abort** button (Return to Top Level ...) and the **Enter Debugger** button (Enter Debugger). If the error was continuable (as `(car 8)` is not) there will also be restarts that allow computation to continue. A restart can be chosen by selecting it (clicking the mouse on it) and clicking the **Invoke Selected Restart** button. Some restarts cause additional dialogs to be displayed, which must be dealt with before computation restarts.

Note that the error was caused by an erroneous call to the function **`car`** but the error message mentions **`rplaca`** and **`rplacd`** rather than **`car`**. This is common in Lisp where functions often call other functions immediately. If you are unsure of the cause of the error, you can use the debugger to find the actual sequence of functions called.

> **The Debugger**. Chapter 6 describes the Debugger, and includes more information about error handling and recovery.

2.9 Loading and Compiling Files

Choosing the menu selections **Load...** or **Compile...** causes a file selection dialog to appear allowing you to select the file or files you wish to load or compile. The file selection dialog a standard MS Windows Common Dialog. We describe its properties in more detail below.

You can also load and/or compile files with the **`load`** and **`compile-file`** functions. Those functions take filename arguments and do not cause a dialog to be displayed.

Loading Files

The **Load...** command on the File menu allows you to read in and evaluate one or more Lisp files. Choose **Load...** and Lisp displays this dialog box:

First select the directory from the widgets on the right. You can specify which file extensions should be shown (by default, only files with extension *.lsp* are shown). Then you can choose one or more files from those listed. Left-click on the file you want to load. The combination Shift-left-click selects all files between one already chosen and the one clicked over. The combination Ctrl-left-click chooses an additional individual file. Left-click also chooses a single file (unselecting all others except the one clicked over). You can also type the name in the File Name box. Click over **OK** to load the selected files and over **Cancel** to abort the load entirely.

Load... uses the **load** function to load the file. **load** may be called recursively. It is both convenient and good practice to divide your program into several files, and load them by loading a single file which itself calls **load** several times to read each of the program files in turn.

Errors while loading

If an error is encountered while loading a file, you have several choices. Consider the following example. Suppose the file *test.lsp* contains the following forms:

```
(defparameter foo 5)
(defparameter bar 10)
(if (car 8) (setq bar 20))   ; Note the invalid form (car 8)
(setq foo 100)
```

Now try to load *test.lsp* by choosing **Load...** from the File menu and specifying *test.lsp*. Loading will proceed until the erroneous form in line 3 --`(car 8)`--is reached. Then Lisp will signal an error about `(car 8)` and provide a Restarts dialog similar to the following:

The error encountered is identified at the top (a bad argument to **car** causes an error in either **rplaca** or **rplacd**). The three restarts have the following effects:

- **Enter the debugger and edit d:\tmp\err.lsp**. Choosing this restart brings up a Debugger window and opens an editor window to the offending file. The cursor will be at the form that caused the error. Note that you *cannot* save this file while the Debugger window is open. You must **Abort** out of the debugger, save the file, and then redo the load. (Some will consider it a misfeature that you cannot save

the file while the debugger is open, but being able to save a file while it is open to the loader will, unfortunately, cause bigger problems.)

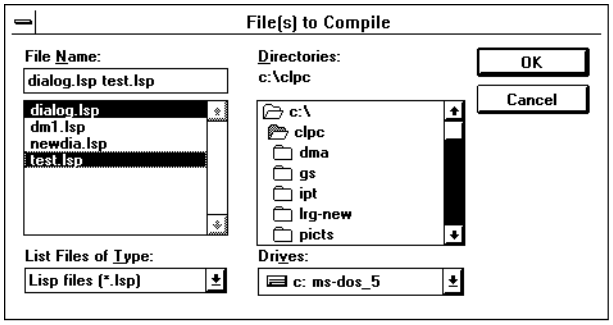
- **Continue loading from #<file-stream ...>**. Choosing this restart causes the loader to skip the offending form and start loading again from the next form. This works in our simple case (when the load completes, foo will be 100 and `bar` will be 5) but note that later forms often depend on earlier forms being evaluated correctly, so this choice can lead to more errors.
- **Return to Top Level (an 'abort' restart)**. The load is cancelled and Lisp returns to its state before the load was started.

The **Abort** button is equivalent to choosing the **Return to Top Level** restart. The **Enter Debugger** button brings up a Debugger window but does not open the file.

Loading is done by **`load`**, which is described in the Online Manual.

Compiling files

Choosing **Compile...** from the File menu brings up a dialog box asking for the files you want to compile, similar to the following:



First select the directory from the widgets on the right. You can specify which file extensions should be shown (by default, only files with extension *.lsp* are shown). You can choose more than one file in the directory to compile. Left-click on the first file and then the combination Shift-left-click selects all files between one already chosen and the one clicked over while the combination Ctrl-left-click chooses an additional individual file (we have selected two in the illustration using Ctrl-left-click). Left-click also chooses a single file (unselecting all others except the one clicked over). You can also type the name in the

File Name box. Click over **OK** to compile the selected files and over **Cancel** to abort the compile entirely. (The Network button, which may appear below the Cancel button but is not illustrated, allows you to add additional directories in which to find files.)

When you click on **OK**, another window appears asking for the name of the compiled file. The appearance of this dialog depends on what kind of Windows you are running. With Windows 95, it looks like the illustration below while with Windows 3.1, it looks like the dialog above. (But only one file can be chosen.) All listed files will be compiled into the one specified file. This window lists other *fsl* (compiled Lisp) files so that you can avoid overwriting an existing file. Note that if you do specify the name of an existing file, Lisp will signal a continuable error asking if you really wish to overwrite the existing file.

Now click **OK** and the files specified will be compiled.

Canceling

At any one of these steps, you can click on the **Cancel** button to abort the process. No files are actually compiled until you click **OK** in the **FSL File to Create** window.

Errors while compiling

If the compiler detects an error while compiling, a Restarts window will be displayed. Sometimes you can continue from the error, in which case the offending form will be corrected or ignored.

2.10 The Window menu: changing and managing windows

The Window menu lists the windows in the current environment. The Manage choice displays a submenu of operations on the currently selected window.

The windows on the menu are divided into two groups, the first being those windows available by default when Lisp is started up and the second being those opened subsequently. A check mark is placed beside the name of each window if changes have been made to the window since it was opened or last saved. Here is the Windows menu in a Lisp session where a Text Edit window to the file *test.lsp* has been opened:

The initial windows. Of the four windows (other than the Toploop) which are initially available, the History and Clipboard windows are described below in this chapter; the Help window is described in Chapter 1; and the Find Definition window is described in section 3.4.

The **Manage** submenu is also displayed. It operates on the selected window. The Second, Third, and Fourth choices bring to the top the second, third, or fourth windows in the occlusion stack (the stack that orders the windows that are displayed).

2.11 Changing Packages

The Packages menu shows all the packages available in the system, with a check mark beside the current one. You can change the current package by choosing a package name from the Packages menu.

Symbol visibility. Changing packages can cause symbols to no longer be visible, so you should exercise care if you switch to another package. Unless you really need to change packages, stay in the `common-lisp-user` package.

The `in-package` function. From within your programs you can change package using **`in-package`**. The current package is held in `*package*`. Packages are described under the heading Packages in the Common Lisp portion of the Online Manual.

Packages in files. Text Edit windows may have a package associated with them. If when you open a file into a Text Edit window, the first form is an in-package form, the current package will be set to that package while that is the active window. The system remembers the package for each window so the current package changes while you move about windows. When the selected window is a Text Editor window, the Packages menu will have a check beside its package.

2.12 The History dialog

The history dialog is a record of the most recent Toploop interactions. When Lisp reads and evaluates a form, it copies the original form to the history list, discarding the oldest interactions when the list becomes too large. The history dialog displays this list. It is displayed by choosing **History** from the Window menu. An example is shown next:

The dialog contains 3 panes:

> **History of Evaluated Forms**, at the top, displays the forms that have been evaluated. It is a scrollable pane and `top:*top-history-limit*` forms are displayed. Clicking on a form in this pane displays it in the Form pane, while double-clicking evaluates it.
>
> **Form**, in the middle, is an editable text widget. You can display a previously-evaluated form there (by clicking on the form in the **History** pane) and edit it, or type a new form directly. Once a form is entered, you can have it evaluated by clicking on the **Evaluate** button or but pressing Enter. To clear the contents of the **Form** pane, select **New History Window Form** from the **Miscellaneous** submenu of the Tools menu. (Ctrl-Alt-N is the keyboard equivalent.)
>
> **Values**, at the bottom, displays the value(s) returned by the selected form in the **History** pane. Only the value(s) of one form can be displayed, but if that form returned multiple values, all are displayed. Selecting a value and clicking on the **Input** button causes the value to be copied to the **Form** pane.

The **Evaluate** button causes the form in the **Form** pane to be evaluated. The **Input** button causes the value selected in the Values pane.

Pressing Tab changes the pane that is selected. Alt-H, Alt-F, and Alt-V selects the specified pane. The arrow keys move about the selected pane.

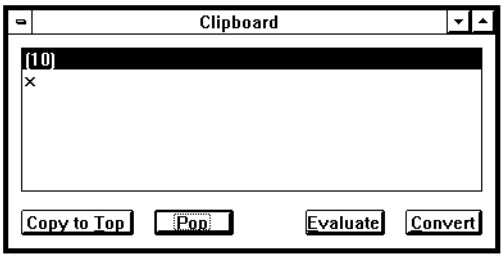
History dialog functions and variables. See section 2.16.3 **The history mechanism** for information on variables and functions associated with the History dialog.

2.13 The Lisp Clipboard

The Windows Operating System has a special area called a Clipboard which can be used to hold a single object, frequently a piece of text. Lisp extends this idea by providing its own Clipboard capable of holding several items at once. The Lisp Clipboard behaves as a stack of Lisp objects which may be manipulated from the Toploop menus. Its principal use is as a place to hold objects while moving them between windows. To display the Clipboard window, choose **Clipboard** from the Windows menu.

Cut and **Copy** operations push the selection onto the Clipboard, discarding the bottom Clipboard item if the total number of items would exceed `*lisp-clipboard-limit*`. The value of this variable can be set in the Toploop preferences dialog box, displayed by choosing **Toploop...** from the Preferences menu. **Paste** operations copy the top item of the Clipboard into the relevant window. They do not affect the contents of the Clipboard.

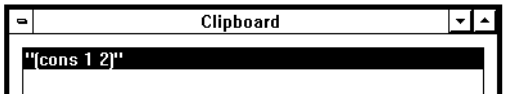
The Clipboard window is a dialog box with four buttons. Here is the Clipboard window after we have chosen Values a couple of times and Input once from the History window displayed above:



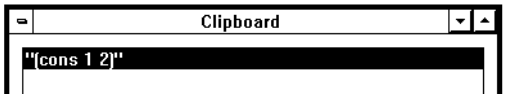
The top of the stack is at the top of the window. As shown, there are four actions possible from the Clipboard window, identified by the four buttons:

- **Copy to Top**. Clicking this causes the selected form to be copied to the top of the Clipboard. (This will cause the form to appear twice in the window.)
- **Pop**. Clicking this removes the top item from the Clipboard stack, discards it and selects the new top item.
- **Evaluate**. Clicking this evaluates the selected item and pushes the first value it returns onto the top of the Clipboard. The value of `x` is 10, so clicking **Evaluate** when the form `x` is selected will cause 10 to appear at the top. Note: not all expressions on the Clipboard can be meaningfully evaluated. Clicking **Evaluate** while `(10)` is selected will signal an error, for example.
- **Convert**. Clicking this takes the current selection, converts it between being a string and being the Lisp expression given by the contents of that string, and puts the result to the top of the Clipboard. We explain this in the next paragraph.

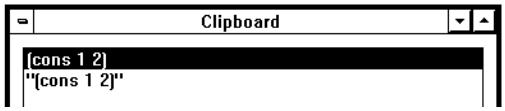
A feature of copying to the Clipboard from a Text Edit window (such as the Toploop window) is that the expression is made into a string. Suppose the form `(cons 1 2)` is selected in the Toploop window:



You can copy it to the Clipboard by choosing **Copy** from the Edit menu but when it is copied, it is converted to a string. Here is the Clipboard window after **Copy** is chosen (we have popped the other items off):



Choosing **Convert** while `"(cons 1 2)"` is selected causes the form `(cons 1 2)` to be placed on the top of the Clipboard. Here is the Clipboard after **Convert** is clicked:



Note that when the selection in the Clipboard is a string, the **Evaluate** button is inoperative (since a string evaluates to itself so evaluation does nothing new). Also, choosing **Convert** when a non-string is selected makes the selection into a string and puts it on top.

Edit menu choices and the Clipboard

As we said above, **Copy** and **Cut** in the Edit menu put things onto the Clipboard. **Pop** in the Edit menu pops the top item on the Clipboard off (just like clicking the **Pop** button). **Paste** copies the top item on the Clipboard to the active window.

The command **Evaluate Clipboard** on the Tools menu may be used from within Editor windows to evaluate the top Clipboard item (regardless of which item is selected, it is the top item that is evaluated). The values returned are printed into the Toploop window and the first value returned is pushed onto the Clipboard stack.

The Windows Operating System Clipboard

Lisp tries to map the top of its Clipboard onto the Windows Operating System Clipboard. On entry to Lisp, the contents of the Windows Operating System Clipboard are pushed onto the Lisp Clipboard as a Lisp string (if possible). On exit from Lisp, the top item on the Lisp Clipboard is copied to the Windows Operating System Clipboard provided that item is a Lisp string. The correspondence between the Lisp and the Windows Operating System Clipboards is maintained while Lisp is running, so you can paste from the Lisp Clipboard in the usual way, provided again that the top item of the Lisp Clipboard is a string. Likewise, cutting or copying from elsewhere puts the selected object into the Lisp Clipboard as a string.

2.14 Setting your preferred Lisp environment

The preferred values of control parameters for Allegro CL for Windows are collectively known as *preferences*. Each control parameter is specified by the value of a particular variable, and may therefore be changed in a program using **`setq`**. The large number of these variables makes them difficult to remember and typing their names is rather tedious, so Lisp makes it easy to change many of them by providing preferences dialogs. Pull down the Preferences menu and there are two choices: Main Preferences and Interface Builder Preferences. The Interface Builder preferences are discussed in the Interface Builder printed manual (in Volume 1). It will not be further discussed here.

Choosing **Main Preferences** displays the following Tab Control dialog:

There are eight forms, each with its own tab. Clicking on a tab displays the associated form. We have displayed the Printer form in the illustration.

There are four buttons at the bottom of the dialog:

Apply. If you have made any changes in any of the preference form, clicking on Apply will cause the current Lisp environment to be updated to reflect your changes. If no changes have been made to any form, this button is masked and clicking over it has no effect. Clicking on **Save** (see below) also applies all changes.

Revert. If you have made any changes to any preference form, but have not applied the changes by clicking on **Apply** (or **Save**), clicking on **Revert** returns all the forms to their state before they were last applied or saved (or initial state if you have never applied or saved them). If no changes have been made or all changes have been applied or saved, this button is masked and clicking over it has no effect.

Save. Clicking on **Save** causes a dialog to be displayed that allows you to choose a file where the preferences will be saved. The default filename and location is *pref.lsp* in the directory where Lisp is installed. We do not recommend that people edit their preferences files (instead, change preferences with the dialog and save the changes, thus overwriting the file). Code at the beginning of the file

checks the version of Allegro CL, and warns you if the version running does not match the version stored.

Close. Closes the Preferences dialog. Unapplied changes will be lost.

The individual widgets on the form can be:

Check-box widgets. Usually associated with boolean variables. Checked means the variable should be t. Unchecked means the variable should be nil. Change by clicking over the box or the variable name.

`Editable-text widgets`. You can place the cursor in the widget (by clicking in it) and then edit as desired.

Combo-boxes (with down arrows to the right). Clicking in the widget or over the arrow displays a menu of choices. Click on a choice to select it or away from the menu to leave the value unchanged.

Font buttons. Short, wide buttons that name a variable associated with a font. Clicking on this button displays a font choice dialog. If a new font is chosen, it becomes the value of the variable (when the references are applied or saved). The button label is printed in the current or chosen font.

The variables associated with the various preferences are documented in this manual (see the index) or in the Online Manual.

2.15 Images

An image is so called because it is a copy, or image, of the Lisp workspace at a particular time, saved onto disk as a large file. If you save an image you can reload it at a later time and continue your work without having to read all your source code in again. Note that images are large: often several Mbytes.

Saving an image

To save an image, choose **Save Image...** from the **Images** submenu of the File menu. Lisp requests a name for the file containing the saved image:

(This is the Windows 3.1 dialog. The Windows 95 dialog looks different but has the same components.) Type a name in the box provided and click **Save**. Lisp closes all the windows on the screen, performs two garbage collections, and saves a copy of the Lisp workspace in the specified file. A new version of the Toploop window is then opened and the session can be continued.

> **Two garbage collections**. Some extra cleaning up of disposable symbols is done between the two garbage collections.
>
> **Image names**. You can give an image file any name you want. We encourage you to add *.img* as the extension to distinguish image files from other files.

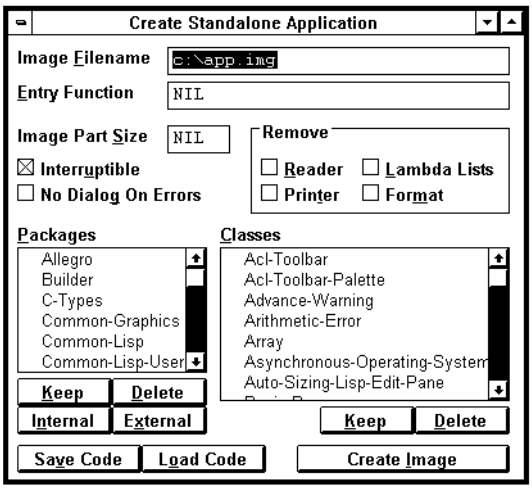
Loading an image

Load Image... is the reverse of **Save Image...**. That is, the whole of the current workspace is replaced by the contents of the loaded image, so the state of the workspace just after loading an image is identical to that which existed when the image was originally saved.

> **Technical note**: the functions associated with these menu choices, **`save-image`** and **`load-image`**, are documented in the Online Manual.

Create Standalone Form

Choosing this item brings up the following dialog:



This dialog is most useful for users who have the Professional version of Allegro CL (and thus have the ability to create runtime images) but it is available in Standard Allegro CL as well. Making the various choices allows you easily to include or leave out features in an image. If you have runtime, then (as described further in the Runtime Generator manual), this form allows you to create a standalone image. Even if you do not have Runtime, you can create images which will include specified functionality.

2.16 Implementation details

This section was chapter 2 of the manual *Inside Programming Tools* in release 2.0.

2.16.1 Starting Allegro CL for Windows

The typical way to start Allegro CL is to double-click over the icon in the Allegro CL program group. However, the actual command line, which can be typed to the dialog displayed by choosing **Run** from the menu displayed by the Start button in Windows 95, or from Program Manager File menu in Windows 3.1, or, to a DOS prompt, is as follows:

```
c:\allegro\lisp.exe c:\allegro\allegro.img
```

Note:

- The directory *c:\allegro* is the default location when installing Allegro CL for Windows. Of course, if you specified a different location, you should use that instead of *c:\allegro*.
- *allegro.img* is the initial image file installed with Allegro CL. You may specify any *.img* file instead of *allegro.img*. In particular, you can specify any *.img* file created with a call to **`save-image`** (or by choosing **Save Image** from the **Images** submenu of the File menu).

2.16.2 Toploop variables and functions

The operations of reading, evaluation and printing are controlled by toploop variables. Their effects are limited to the toploop. For example, rebinding `*top-print-length*` only affects output from the toploop; conversely, rebinding `*print-length*` has no effect on toploop printing.

The bindings of `top:*top-read*`, `top:*top-eval*` and `top:*top-print*` must be suitable as the first argument of **`funcall`**.

> **Note: many symbols described in this chapter are in the `toploop` package. This package is not used by default by the `common-lisp-user` package and**

is not in the default use list of `in-package` or `make-package`. Therefore, when you refer to these symbols, you must qualify them (with `toploop:` or `top:`) or use the `toploop` package with `use-package`.

`*toploop-window*` [Variable]

Package: `toploop`

■ The value of this variable is the toploop frame window, whose single child pane is the stream which is the value of `*terminal-io*`.

`*top-print-level*` [Variable]

Package: `allegro`

■ controls the maximum depth to which results of evaluation of input are printed by the toploop. Its initial value is 4.

`*top-print-length*` [Variable]

Package: `allegro`

■ controls the maximum length to which results of evaluation are printed by the toploop. Its initial value is 8.

`*top-read*` [Variable]

Package: `toploop`

■ is bound to the function used by the toploop for reading.

`*top-eval*` [Variable]

Package: `toploop`

■ is bound to the function used by the toploop for evaluating input.

`*top-print*` [Variable]

Package: `toploop`

■ is bound to the function used by the toploop for printing.

`*top-prompt*` [Variable]

Package: `toploop`

■ is a **`format`** string used to print the toploop prompt. **`format`** is called with five arguments:

```
(format t top:*top-prompt*
```

transaction-number top:*top-level* nil)

`transaction-number` is an integer which is incremented each time the user is prompted for input. `nil` is included in the argument list to ensure correct operation of ~V. The default value of `top:*top-prompt*` is "~%~*~V@{>~}".

`*top-level*` [Variable]

Package: `toploop`

- is an integer giving the depth of the current toploop invocation.

`*top-query-exit*` [Variable]

Package: `toploop`

- if this variable is non-`nil`, the user is prompted for confirmation when an attempt is made to quit the Lisp application.

`*toploop-comtab*` [Variable]

Package: `toploop`

- is the comtab used by the toploop.

`find-toploop-prompt` [Function]

Arguments:

Package: `toploop`

- Calling this function causes the toploop window to be exposed, a new Lisp prompt to be generated, and the cursor to move to the position after the new cursor. Calling this function is a quick and easy way to get to the main Lisp prompt. It is also convenient when the prompt has been deleted for some reason.

- Each of the built-in comtabs binds the key combination Control-Shift-N to this function, so that key combination will find a fresh Lisp prompt whenever the input focus is in any text window (including the toploop, file editors, and the Help window).

2.16.3 History mechanism

The toploop retains a history of every form input to it, and the results of evaluation. Entries in the history list can be inspected, edited and re-evaluated. An interface is provided so that user programs may also use the history facility.

`*top-history-list*` [Variable]

Package: `toploop`

- holds the toploop history list.

`*top-history-limit*` [Variable]

Package: `toploop`

- controls the number of entries to be retained in the history list. Its initial value is 25.

`*top-history-count*` [Variable]

Package: `toploop`

- holds the number of the current interaction with the toploop.

`top-push-history` [Function]

Arguments: *`item`*

Package: `toploop`

- pushes the given history item *`item`* onto the history list, and trims the history list to `*top-history-limit*` items. A history item is a cons whose car is an input form and whose cdr is a list of the values returned by evaluating the input form.

`top-replace-history` [Function]

Arguments: *`item`* `&optional` *`(n -1)`*

Package: `toploop`

- replaces the *n*th item in the history list by *`item`*. *`n`* must be an integer. If it is positive, *`item`* will replace that at the *n*th position in the history. If it is negative, *`item`* will replace that at the *n*th position back in the history. *`item`* should be a cons whose car is an input form and whose cdr is a list of values returned by evaluating that form.

2.16.4 Finding source code

The toploop keeps track of the location of definitions evaluated, loaded, or in opened text files. This is for use by the **Find Definition** menu command.

`find-symbol-definition` [Function]

Arguments: *`symbol`*

Package: `toploop`

- attempts to find source code for symbol.

`delete-definitions` [Function]

Arguments: *`symbol`*

Package: `toploop`

- removes all saved definition information for *`symbol`*.

`find-method-definition` [Function]

Arguments: *`method`*

Package: `toploop`

- attempts to find source code for *`method`*.

`find-method-definition-from-name` [Function]

Arguments: *`generic-function-name specializer-class-names`* `&optional` *`qualifiers`*

Package: `toploop`

- Finds source code for a particular method from the name of the generic function, a list of the names of the method's specializers, and optionally a qualifier such as `:before`, `:after` or `:around`.
- Example:

```
(top:find-method-definition-from-name
  'resize-window '(status-bar t) :after)
```

`find-applicable-method-definitions` [Function]

Arguments: *`generic-function-name arguments`*

Package: `toploop`

■ Locates source code for all methods of *generic-function-name* that are applicable to a list of *arguments* and that were loaded or evaluated from source code. If exactly one applicable method is found, then the source is displayed immediately in an editor window. If more than one definition is available, then they are listed in the Find Definition dialog.

■ Example:

```
(find-applicable-method-definitions
           'redisplay-window (list top::*toploop-window* t))
```

defdefiner [Macro]

Arguments: *symbol type*
&optional *(full-name-extractor* 'second*)*

Package: toploop

■ marks *symbol* as a macro which creates a definition of the specified *type*. *full-name-extractor* is a function that can be applied to the defining form to extract a name for the definition. Whenever a top level call to **read** reads such a definition, the file or window it was read from is attached to the name. A later **find-symbol-definition** for the name will look in the appropriate file or window. Standard symbols such as **defun**, **defmacro** etc. have already been set up to do this by **defdefiner**.

■ For example:

```
(defmacro mydefun (&body body)
    `(progn
        (print 'mydefun)
        (defun ,@body)))
```

defines **mydefun** and `(defdefiner mydefun myfun)` treats **mydefun** as a definer of type `myfun`, so that `(mydefun ploppi (x) (+ 8 x))` is a definition of **ploppi**.

■ Source code examples:

```
(defdefiner defclass class)
(defdefiner defsetf setf
        (lambda (form)
            (list 'setf (second form))))
```

definer-p [Function]

Arguments: *symbol*

Package: toploop

- returns definer type of *symbol* or nil.

2.16.5 Writing a toploop

The function **toploop** allows the user to install his or her own form of toploop.

toploop [Function]

Arguments: &key (:read-fn *top-read*)
(:eval-fn *top-eval*)
(:print-fn *top-print*)
(:prompt *top-prompt*)

Package: toploop

- is called to establish a new toploop, which uses *standard-input* and *standard-output* for input and output. Using the given functions for reading, evaluating and printing input, a **read-eval-print** loop is then invoked. The toploop prompt is constructed from *prompt*.

2.16.6 The status bar

The status bar is a static text dialog that appears at the bottom of the screen when Allegro CL for Windows is the current application. The following function and two variables are associated with the status bar.

lisp-message [Function]

Arguments: *format-string* &rest *format-args*

Package: common-graphics

■ This functions prints (using **format**) *format-string* to the status bar, using *format-args* as required. It is not an error to call this function when the status bar does not exist or is hidden, but nothing will be printed in that case. (The associated function **cg:window-message** prints to the status bar in any window. See the description of that function in the *Common Graphics* portion of the Online Manual.)

`*lisp-message-print-length*` [Variable]

Package: common-graphics

■ This variable controls the length of lists printed in the status bar. Like `cl:*print-length*`, its value can be a positive integer or `nil`. If the value is a positive integer, no more than that many items will be printed in a list, with the presence additional items indicated by suspension points (...). If the value is `nil`, all items are printed.

`*lisp-message-print-level*` [Variable]

Package: common-graphics

■ This variable controls the depth of lists printed in the status bar. Like `cl:*print-level*`, its value can be a positive integer or `nil`. If the value is a positive integer, no more than that many levels will be printed in a list, with the presence additional levels indicated by suspension points (...). If the value is `nil`, all levels are printed. (If a list contains no sublists, it has depth 1. A list containing at least one sublist of depth `n` and no sublist of depth greater than `n` has depth `n+1`. If the value of this variable is 1, all sublists will be indicated with #.)

`*lisp-status-bar-font*` [Variable]

Package: toploop

■ The font used in the Lisp status bar.

`*lisp-status-bar-number-of-lines*` [Variable]

Package: toploop

■ The number of lines in the Lisp status bar.

Chapter 3 The Text Editor

Using the Text Editor, you can edit files containing Lisp programs from within the Lisp application. The Text Editor is similar to the Windows Notepad Editor, but with additional features to make the editing of Lisp programs very easy. The Text Editor also has a mode of working which make it resemble Emacs.

3.1 Opening a file

Before a file can be edited, you need to open that file. You can work on a new file or one that you've worked on and saved before. You can have several files open at one time.

Creating (opening) a new file

Choose **New** from the File menu. Lisp displays a new, empty window with a title such as "Untitled.1". As you create successive new files, the number in the window title will be incremented to give each file a unique name.

Opening an existing file

Choose **Open...** from the File menu. Lisp displays a Common Dialog for choosing a file-name. The appearance of the dialog depends on the type of Windows you are running (the Windows 95 dialog looks quite different from the Windows 3.1 dialog) but either one allows you to specify a drive, directory, and filename.

After you have specified the file clicked **Open** or **OK** (depending on the dialog) Lisp displays a new window containing the file you have selected. The window title is set to the file name.

> **Active files**. When you have more than one file open, only one is active. The active file window is typically on top of the other windows and its insertion point

is typically flashing. To work with a different file, you must make it the active file by clicking in its window or by choosing its title from the Windows menu.

File types. Files of any type can be opened but only text files can be meaningfully worked on.

32Kbyte Size limitation. Note that there is a 32Kbyte limit on the size of files opened in a Text Editor window. (Larger files can be compiled and loaded. They just cannot be edited with the Text Editor. 32Kbytes is typically between 500 and 1000 lines of code.)

3.2 Saving A file

When you are working with a file, you are really working with a temporary copy of it. To make a permanent record of the file, you must save it to disk.

When the contents of a text-edit-window need to be saved, an asterisk (*) appears in the title bar. To turn this feature off, see the description of `te:*flag-modified-text-window-p*` in section 3.18.18 below. Note that the asterisk is not part of the title string.

Saving for the first time

Files with names like *Untitled.3* have never been saved to disk. To save them, choose **Save As...** or **Save** from the File menu. Lisp displays a common dialog titled Save As and asks for the file name. The appearance of the dialog depends on the type of Windows (the WIndows 95 dialog looks different from the Windows 3.1 dialog) but both allow you to choose a drive and directory and provides a widget to type in the filename. Click on **OK** or **Save** (depending on the type of dialog) and the file will be saved.

Saving again

If you have already saved a file, you can save it again using the same name. Choose **Save** from the File menu. Lisp replaces the previously saved version of the file with the new version.

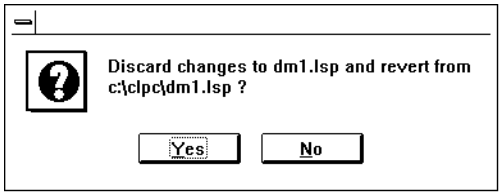
Saving under a different name

If you make changes in a file and want to save the original version and the new version, you must save the new version under a different name. Choose **Save As...** from the File menu. Type the new file name and click **Save**. The window's title is changed to the new file name. You can also use this procedure to make an exact copy of an existing file.

> **Contents of files**. Files handled by Lisp do not have to contain valid Lisp programs: in fact, they can contain any text you like.
>
> **Unsaved changes**. Files in which changes have been made but not saved are shown with a check mark next to their names in the Windows menu.

Discarding changes

You can cancel any changes in a file since you last saved it. Choose **Revert to Saved** from the File menu. Lisp asks you to confirm your decision:



Click **Yes** to discard the changes. Clicking **No** cancels the Revert to Saved command and leaves the file unchanged.

> **Warning**. The **Revert to Saved** command cannot be undone. Any changes you have made to the file will be lost.

3.3 Closing a File

Closing a file removes it from the screen and gives you the chance to save any unsaved changes to disk. Choose **Close** from the File menu or from the Manage submenu of the Window menu -- these two **Close** menu items are the same). Alternatively, you can double click in the file's Control-menu box. If there are unsaved changes, Lisp asks if you want to save the file:

Click **Save** to save the changes before closing the file. Click **Discard** to close the file without saving changes. Click **Cancel** to stop the file being closed.

> **Exiting without saving**. If you choose **Exit** from the File menu to exit Lisp (or attempt to exit Lisp in any other way, such as trying to close the final Toploop window) while a file you have changed is open, Lisp asks you whether you want to save it. So you don't have to worry about closing all your files before exiting Lisp.

3.4 Finding a definition

Allegro CL for Windows keeps a record of where objects (functions, variables, macros, etc.) were defined. When a symbol is selected or the cursor is near a symbol, choosing **Find Definition** from the Search menu will display the window where the definition of the object named by the symbol appears (for example, the toploop window if the object was defined there, a window open to a file if the definition was there, etc.) If no source for the definition is known, that fact is printed to the status bar.

Sometimes there is more than one object named by the symbol. Suppose, for example, that we have defined a function **`foo`** and a global variable `foo`. We defined the function in the toploop window as follows:

```
(defun foo (x) (+ 1 x))
```

We defined the variable in a window open to the file *dm.lsp* as follows:

```
(defvar foo 10)
```

Now, there are two definitions to choose from and Lisp brings up the following dialog to allow you to specify the definition you want:

The All button

In some cases (but not in the illustrated case) only some effective method definition are displayed. Clicking on **All** will display all such definitions. If not applicable (as in our example), the button is masked.

The Edit button

Clicking on the **Edit** button finds the source code of the selected definition. If the definition is in an open Text Edit window, that window is made active and scrolled to the definition. (The Toploop window is a Text Edit window so the last statement applies. In the example above, clicking on **Edit** would scroll the Toploop window to the definition of `foo`.)

If the definition came from a file loaded with load, clicking on **Edit** opens a Text Edit window to the file (the source file if a compiled file was loaded).

Defining you own definition

As delivered, Allegro CL for Windows knows about the standard Lisp definition forms (`defun`, `defmacro`, `defvar`, etc.) and will find all definitions made by such forms. You may have extended Lisp with new macros that define objects. You can have Lisp find those definitions as well with the macro `defdefiner` described in section 2.16.4 **Finding Source Code** earlier in this manual.

Other things to note

- If the definitions for one symbol are listed and you enter a new symbol in the **Symbol** box, the window is typically updated automatically. However, it is possible for the updating to be incomplete, particularly if the new symbol's name simple adds letter to the old symbol name (the symbol `foo1` compared to the symbol `foo`). Clicking the **List** box guarantees the updating is complete.
- If you delete the file where the definition is stored or modify it by removing the definition and then click on **Edit**, Lisp will obviously not find the definition (since it no longer exists). If will open or try to open the file, however.

Redefinition warnings

Because Lisp remembers where something was defined, it is able to warn you if you redefine something in another location, since the redefinition may be unintentional. You will be warned about redefinitions if the `*warn-on-redefinitions*` box is checked in the Text Editor preferences form of the Preferences dialog:

(The box title names the associated variable, which is in the `allegro` package -- see the description in the Online Manual for more information.) If redefinition warning is enabled (it is in our case, since the box is checked), here is what happens. Suppose we have defined the variable `myfun` in the file *dm1.lsp* with the form:

```
(defvar myfun 10)
```

Suppose we now try to evaluate the form `(defvar myfun 0)` in the toploop window. An error is signaled and the restarts window appears with the following restarts available:

> **Redefine it**. Choosing this restart causes the new definition (in the new location) to b replace the existing definition.
>
> **Return to Top Level (an 'abort' restart)**. Cancels the loading of the file containing the redefinition (if it comes from a file) or just cancels the redefinition (if it was typed directly to Lisp).

The following restart is only available if the redefinition appears in a file being loaded:

> **Continue loading from #<file stream ...>**. The form causing the redefinition is skipped and the remaining forms are loaded.

The final restart is only available if the redefinition appears in a source (not compiled) file being loaded:

> **Enter debugger and edit <file>**. This brings up a debugger window (not of much interest since the cause of the error is presumably clear) and opens the file for editing. You must abort out of the debugger before you can save any changes to the file.

Redefining a system or Lisp function or object

If you try to redefine a system standard Lisp object (like the function **`car`**), a different error will be signaled. This behavior is controlled by the variables `*warn-on-protected-class-definition*`, `*warn-on-protected-function-redefinition*`, and `*warn-on-protected-generic-function-redefinition*`, all in the `allegro` package and all defined in the Online Manual. You could continue from the error and effect the redefinition but we *strongly* advise you not to redefine system or standard Lisp functions. You may break the system in weird and unrecoverable ways. These variables are set on the Compiler preferences form but we recommend that they be kept true, which is their initial values.

3.5 Inserting Text

To insert text in a file, you type it at the insertion point, which appears as a blinking vertical bar. As you type a closing parenthesis, Lisp flashes the corresponding opening parenthesis to allow you to check the bracketing of your code.

Pressing Enter moves the insertion point to a new line and indents it appropriately. Pressing Tab when the insertion point is at the beginning of a line reindents the current line in relation to the line above it.

Overriding indentation. You can use the arrow keys to override indentation. Press ← as often as necessary to move toward the beginning of the line.

Moving the insertion point

You can move the insertion point by moving the cursor to where you want to insert the text. (the pointer appears as an I-beam in a Text Editor window). Click the location. Lisp moves the insertion point to the new location. If the text you want to change isn't visible in the file window, you can move around in the file using the scroll bars. You can also drag the window around the screen, resize it and zoom it.

Shortcuts. You can use the arrows keys to move the insertion point in a file.

Selecting text

You must select text before you can perform a command on it, such as copying or moving. Selected text can be any size from a single character up to the entire file. When you select text, Lisp highlights it.

You can select text by dragging through it. You can also select text by clicking a location and then Shift-clicking in a different location. Lisp selects all the text between the two locations.

Selecting a word. You can select a word by double-clicking anywhere in it. Lisp defines a word as a complete Common Lisp symbol or number.

Selecting a Lisp form. You can double-click on an opening or closing parenthesis or square bracket to select the whole of the expression up to the matching parenthesis or bracket. This is useful to check that a program is bracketed properly.

Selecting a string. You can double-click on a string quote (") to select all the text forward to the next string quote.

Selecting a comment. You can double-click on a semicolon to select the rest of the line including the semicolon. You can double-click on a vertical bar (|) to select all the text forward to the next vertical bar.

Selecting the entire file. To select the entire file, choose **Select All** from the Edit menu.

3.6 Deleting Text

To delete one character at a time, click the location to the right of the character you want to delete and press the Backspace key. Continue to press Backspace to delete more characters. Or position the cursor to the left of the characters you want to delete and press the Delete key (which deletes forward).

To delete a block of text, select the text you want to delete and choose **Delete** from the Edit menu. Alternatively, you can press the Delete key or the Backspace key (either deletes all selected text).

To replace a block of text, select it and start typing. The newly typed text replaces the old text.

Transposed characters. One of the most common typing mistakes is to transpose characters: for example, typing *defnu* instead of *defun*. In the Host Text Editor mode, you can type Ctrl-‘ to switch the positions of the two characters to the left of the insertion point.

Shortcuts. In the Host Text Editor mode, you can type Ctrl-Delete to delete the word after the cursor, Ctrl-Backspace to delete the word before the cursor, and Alt-Backspace to delete the last form you typed.

3.7 Moving and Copying Text

You can move text by cutting it and then pasting it in a new location. You can copy text to used the same text in different places without retyping it. You can move or copy text within the same file, or to any other file.

Select the text you want to move or copy and choose **Cut** or **Copy** from the Edit menu. Lisp puts the selected text into the Clipboard. If you **Cut**, the selected text is deleted from the file. Click the location where you want the text to appear and choose **Paste** from the Edit menu. Lisp copies the text from the Clipboard and inserts it at the new location.

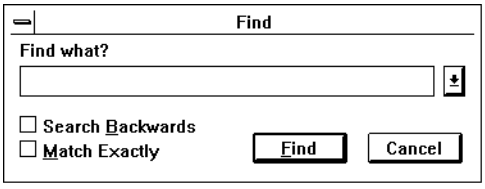
Pasting onto a selection. If you select some text before choosing the **Paste** command, that text will be replaced with the text from the Clipboard.

Lisp uses a special Clipboard. When you copy or cut text into the Lisp Clipboard, it does not replace what was already there. Instead, the text is pushed onto the Lisp Clipboard (see section 2.13). You can choose **Clipboard** from the Windows menu to see the contents of the Lisp Clipboard.

Moving and copying to the Structure Editor. You can move or copy things between Text Editor windows and Structure Editor windows. When moving text to the Structure Editor, you may need to use the **Convert** button on the Clipboard window to convert your text to Lisp structure before pasting it in.

3.8 Finding Text

You can easily locate any text in a file. Click the location in the file where you want the search to begin. Choose **Find...** from the Search menu. Lisp asks for the text to search for:



Type the characters you want to find in the space provided. As displayed, the search will be forward. Click **Search Backwards** if you want to search backward from the insertion point (an x appears in the box to the left of **Search Backwards**). Click again to remove the x and search forward. Similarly, click **Match Exactly** if you want the search to be case sensitive. Then click **Find**. Lisp selects the next occurrence of the characters you have typed. If the characters cannot be found, the system beeps.

Case sensitive searching. When the **Match Exactly** box is not checked, Lisp counts lowercase and uppercase characters as being the same. In addition, it counts ′ as being the same as ", (the same as [, and) the same as].

Special characters. To search for a Tab character in a file, type \T in the Find box. To search for an end of line, type \N (“Newline”). To search for the Backslash character itself, type \\.

Repeating a Find command

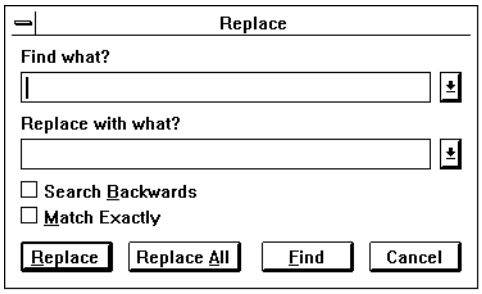
To search from another occurrence of the characters you typed into the Find What box, choose **Find Again** from the Search menu. The search will continue in the same direction as before.

Searching using the Clipboard

To search a file for the text in the top item of the Lisp Clipboard, choose **Find Clipboard** from the Search menu. The search direction and case sensitivity are taken from the values used in the most recent **Find...** command. You will often find it easier to select some text and then use the **Copy** and **Find Clipboard** commands than to use the **Find...** command.

3.9 Replacing Text

Lisp lets you replace any text in a file with some other text. Click the location in the file where you want the search to begin. Choose **Replace...** from the Search menu. Lisp displays a Replace dialog box which asks for the text to search for, and the text to replace it with:



Type the characters you want to find in the **Find what?** box. Type the new characters in the **Replace with what?** box. Click **Search Backward** if you want to search backward from the insertion point. Click **Match Exactly** if you want the search to be case sensitive. Then click **Replace**. Lisp replaces the next occurrence of the **Find what?** characters with the **Replace with what?** characters and selects the new characters. If the **Find what?** characters cannot be found, the system beeps.

Repeating a Replace command

To repeat the last **Replace** command, choose **Replace Again** from the Search menu. The search for a second occurrence of the Find characters continues in the same direction as before.

Replacing all occurrences of some text

If you click **Replace All** on the Replace dialog box, all occurrences of the **Find What?** characters are replaced between the insertion point and the beginning or end of the file, depending on the search direction.

Replacing text selectively

You can replace some, but not all, occurrences of the **Find** characters in a file. Choose **Replace...** from the Search menu and proceed as before but click the **Find** button instead of the **Replace** button. Lisp selects the next occurrence of the **Find** characters. Now choose **Replace Again** from the Search menu to replace that occurrence, or choose **Find Again** to find the next occurrence. Continue using **Find Again** and **Replace Again** until you have replaced all the occurrences of the Find characters that you wish to change.

3.10 Marking Text

Lisp lets you mark a location in a file so that you can easily return to that place later. You can also select all the text between the marked location and another location in the file.

Marking a location

Click the location in the file where you want to put the mark and choose **Set Mark** from the Marks submenu of the Search menu. Lisp remembers the location which you have marked. You can only have one mark in each file. If you mark a different location, the previous mark is lost.

Moving to the mark

You can move the insertion point to the location you have marked by choosing **Swap with Mark** from the Marks submenu of the Search menu. Lisp swaps the locations of the mark and the insertion point.

Selecting text using the mark

Choose **Select to Mark** from the Search menu. Lisp selects all the text between the mark and the insertion point. The mark is unaffected. Choosing **Display Selection** from the same menu moves scrolls the window so that the mark is visible.

Finding out where the mark is

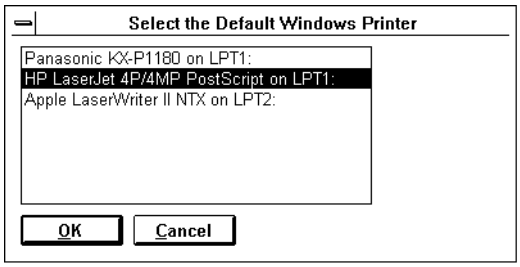
You can quickly see where the mark is set by using the **Swap with Mark** command twice. The first time you use it, Lisp moves the insertion point to the mark. The second time, Lisp moves the insertion point back to its original position.

3.11 Printing

You can produce a printed copy of any Lisp file or part of a file. You can specify page and printing settings so that the printed copy appears in the format you want.

Choosing a printer

Choosing **Choose Printer...** from the Print submenu of the File menu displays a dialog, like that shown below, containing the list of installed Windows printers. If you select a printer from this dialog and click OK, then Allegro CL modifies the *win.ini* file to make the chosen printer the current default printer for all Windows applications (just as if you selected it using the Windows Control Panel). Allegro CL will now use that printer as well.



Setting up the page format

You can specify options appropriate to the printer by choosing **Printer Setup...** from the File menu. Lisp uses the default printer specified by bringing up the Control Panel application and selecting Printers. If you wish to use an alternate printer, change the default before asking Lisp to print. Once things are set up, you can print.

Printing a file

To print a copy of a file open to a Text Edit window, choose **Print...** from the Print submenu of the File menu while the window is active. Depending on the printer, Lisp may display another dialog box asking for more information or just start printing (as it does with the printer shown in the illustration).

Printing part of a file. To print part of a file, select the text you want to print and then choose **Print...** from the File menu.

Finding out if there is selected text in a file. You can find out whether there is a selection in a file by choosing **Display Selection** from the Search menu. Lisp scrolls the file so that the insertion point (or the start of the selection) is visible.

3.12 Evaluating Lisp Forms

You can evaluate Lisp forms in any file. The values returned by the evaluation are printed in the Toploop window and recorded on the history list, exactly as if you had typed the text into the Toploop window yourself.

Evaluating single forms

To evaluate a single form in a file, click the location in the file just before the form to be evaluated and press Alt-Enter. Lisp reads from the file starting at the insertion point and evaluates what it has read. The insertion point is left just after the form that has been read.

Evaluating many forms

To evaluate more than one form at a time, select the forms you want to evaluate and choose **Selection** from the submenu displayed by choosing **Evaluate** from the Tools menu. Lisp reads the forms in the selection one at a time and evaluates them.

Repeatedly evaluating one form

After pressing Enter, you can move the insertion point back to the start of the form just evaluated using Alt← in the Host text editor mode.

Worksheets. You can create a file containing forms that you need to evaluate frequently and keep it open on the screen. When you need to evaluate one of the forms in it, you can easily find the form in the file, click before it and press Alt-Enter.

Stepping through a file. You can step through a file, evaluating one form at a time, by clicking at the start of the file and repeatedly pressing Enter.

Evaluating an entire file. To evaluate all the forms in a file at once, choose **Select All** from the Edit menu, then **Selection** from the submenu displayed by choosing **Evaluate** from the Tools menu. Alternatively, you can save the file to disk and then use the **Load...** command from the File menu.

Selections that contain incomplete forms

The **Evaluate:Selection** command works by repeatedly reading forms until it has read beyond the end of the selection. If your selection does not end at the end of a form, then **Evaluate:Selection** may read past the end of the selection (it will try closing any open parentheses). If it reads off the end of the file, Lisp signals an error, perhaps because the form closed off with extra parentheses is erroneous, perhaps for reading off the end of the stream.

3.13 Reformatting a File

When you have edited and re-edited a file containing Lisp programs, the layout can often become untidy and the indentation misleading. Lisp allows you to reformat part (or all) of a file to check parenthesis errors and improve its presentation.

Setting up

Lisp gives you two ways of reformatting a file. You can tell Lisp to reformat by reading from the file and then pretty print back what it has read using the Lisp Printer. Alternatively, you can tell Lisp to reformat by re-indenting each line of the file with respect to the bracketing of the previous line. To set your reformatting preference, bring up the Preferences dialog (by choosing **Main Preferences** from the Preferences menu) and choose the Text Editor

tab so the Text Editor preferences form is displayed. The `*pretty-printer*` field has a Combo box next to it, with the choices Reindent and Pretty Print. If the value shown is not what you want, change it and click on the **Apply** button.

Reformatting

Select the form(s) you want to reformat and choose **Pretty Print** from the Tools menu. Lisp reformats the selection. To reformat the entire file, choose **Select All** from the Edit menu before using the **Pretty Print** command.

> **Reformatting comments**. When reformatting using the pretty printer, Lisp attempts to justify comments spanning more than one line. Lisp justifies comments into left-justified paragraphs. A paragraph is considered to end when a blank comment line is found, or when a comment line beginning with two or more spaces is found. You can turn justification of comments off by unchecking the `*justify-comments*` box in the Text Editor from on the Preferences dialog (displayed by choosing **Main Preferences** from the Preferences menu and tabbing to Text Editor).
>
> **Changing the case of reformatted text**. The Lisp printer prints symbols in uppercase or lowercase according to the value of `*print-case*`. You can change the case of a reformatted selection by choosing **Change Case** from the Edit menu. Lisp displays the submenu with choices **UPCASE**, **downcase**, and **Capitalize**. Click on the type of casing you want and Lisp changes the case of all the text in the selection

Comments

You can easily comment out sections of a program using Lisp. You can then uncomment them back in again later.

Commenting out part of a file

Select the form(s) you want to comment out and choose **Comment In/Out** from the Edit menu. Lisp inserts three semicolons before each line in the selection. To uncomment the selection simply use the **Comment In/Out** command again.

> **Determining whether the selection will be commented in or out**. Lisp only looks at the first line of the selection, to determine whether to add semicolons or remove them.

Single semicolon comments. In the normal text editor mode, pressing Ctrl-; tabs towards the right of the file and inserts a single semicolon. You can change the amount of whitespace tabbed by modifying the `*comment-indent*` field in the Text Editor from on the Preferences dialog (displayed by choosing **Main Preferences** from the Preferences menu and tabbing to Text Editor. This also changes the tabulation used by the Lisp printer.

3.14 Associating Packages With Text Editor Windows

In nearly all cases, the forms saved in a file will need to be read into a particular package. To make sure that the forms are read into the right package when the file is loaded, you should normally put a call to **`in-package`** at the start of the file.

When you open a Text Edit window to a file with an **`in-package`** form, the current package is set to that package while the Text Edit window is active. Each Text edit window has its own associated package so the current package changes as you move among Text Edit windows.

Checking the package of a file

You can quickly check the packages associated with a file by looking at the Packages menu while that file is active. The package into which forms in the file will be read is shown with a check mark.

3.15 Setting The Text Editor Mode

You can alter the Text Editor so that it behaves like an Emacs or Brief editor (rather than a standard Windows Operating System editor -- called Host in the menu -- which is the default). You may find this useful if you have used one of these other editors before.

Altering the mode

To alter the Text Editor mode, change the value in the `*editor-mode*` box in the Text Editor from on the Preferences dialog (displayed by choosing **Main Preferences** from the Preferences menu and tabbing to Text Editor. Choose the desired mode and click on **Apply**. Lisp now sets the Text Editor to work in the new mode in all the open files.

Keybindings in editor modes

Appendix A has a table of keybindings in effect in the various editor modes.

Using editor commands from any mode

Changing the editor mode just affects the editor commands that are available from the keyboard.

Menu bar menu shortcuts. In Emacs mode, menu bar menu shortcuts using the Alt key and the underlined letter in the menu are not available. You can still choose commands from the menu in the normal way.

The Toploop window. Changing the Text Editor mode also affects the text editing commands available in the Toploop window, since Toploop window is a Text Editor window.

Defining editor commands. You can define your own editor commands.You can also bind your commands to keys. See section 3.17 and its subsections below for more information.

Shortcuts

In the Host Text editor mode, the following keyboard shortcuts work (some but not all work in other modes):

← moves the insertion point back one character.

→ moves the insertion point forward one character.

↑ moves the insertion point up one line.

↓ moves the insertion point down one line.

Ctrl ← moves the insertion point back to the start of the enclosing list.

Ctrl → moves the insertion point forward to the end of the enclosing list.

Ctrl ↑ moves the insertion point back to the start of a definition.

Ctrl ↓ moves the insertion point to the start of the next definition.

Alt ← moves the insertion point back one form.

Alt → moves the insertion point forward one form.

Alt ↑ moves the insertion point to the beginning of the current line.

Alt ↓ moves the insertion point to the end of the current line.

Delete deletes the character to the right of the insertion point (i.e. deletes forward).

Backspace deletes the character to the left of the insertion point (i.e. deletes backward).

Alt-Delete deletes all the characters back from the insertion point to the beginning of the line.

Ctrl-Delete deletes the form to the left of the insertion point.

Enter inserts a #\NEWLINE character and indents the new line according to the bracketing of the previous line.

Tab at the beginning of a line, reindents the current line according to the bracketing of the previous line. If the insertion point is not at the beginning of a line, pressing Tab inserts three spaces.

Ctrl-; tabs over and inserts a single semicolon.

Ctrl-’ transposes the two characters to the left of the insertion point.

Ctrl-Space selects all the text between the insertion point and the mark.

3.16 Text editor internals

This material was chapter 3 of the *Inside Programming Tools* manual in release 2.0.

The Allegro CL for Windows Text Editor is an Emacs-like programmable Text Editor. Any stream opened with a *device* argument of `text-edit-pane` supports all the Text Editor functions described in this chapter. In addition, `lisp-edit-panes` (subclasses of `text-edit-pane` and `comtab-mixin`) support parenthesis flashing and command tables (comtabs). `text-edit-pane`s are based on the Edit controls of version 3.x of the Windows operating system and as such are limited to 32K characters.

Note that the pane of a window is a child of a frame window and it is common to have a handle on the frame rather than the pane. If the window has no other children, the pane is the single element of the list returned by applying **`windows`** to the window. Most functions in this chapter operate on panes and will fail if passed a frame window rather than a pane as an argument.

A special event function associated with Lisp Edit windows turns events such as mouse clicks and key presses into calls to Lisp functions by looking up the events in command tables or comtabs. The functions used to set up the comtabs themselves are described in Chapter 9 of this manual. This chapter describes the functions which you will need if you want to extend the standard Text Editor or write text processing functions.

Functions are provided to manipulate Lisp programs as characters, lines, forms, lists, complete definitions and selected sections of text. The Text Editor only has access to the character representation of a program, unlike the Structure Editor which manipulates Lisp programs as data structures.

3.16.1 Operations on characters

`forward-character` `[Function]`

Arguments: *pane*

Package: `text-edit`

- increments the file position by one and returns the character stepped over. **`forward-character`** returns `nil` if the file position is at the end of *pane*.

backward-character [Function]

Arguments: *pane*

Package: text-edit

■ decrements the file position by one and returns the character stepped over. **backward-character** returns `nil` if the file position is at the start of *pane*.

delete-next-character [Function]

Arguments: *pane*

Package: text-edit

■ deletes the character to the right of the file position. Does nothing if the current file position is at the end of *pane*.

delete-previous-character [Function]

Arguments: *pane*

Package: text-edit

■ deletes the character to the left of the file position and decrements the file position by one. Does nothing if the file position is at the start of *pane*.

insert-character [Function]

Arguments: *pane character*

Package: text-edit

■ inserts *character* at the current file position and increments the position by one.

transpose-characters [Function]

Arguments: *pane*

Package: text-edit

■ transposes the two characters to the left of the file position and leaves the file position unchanged.

3.16.2 Operations on words

The functions described in this section operate on whole words. A word is a sequence of constituent characters delimited by characters with syntax attributes terminating-macro, whitespace, super-bracket or illegal in the current readtable (which is the value of `*readtable*`).

The file position is inside a word if the characters to its left and right have the character syntax constituent.

If the file position is inside a word, the next or current word is that word. Otherwise it is the next sequence of constituent characters found in the file.

Similarly, the previous word is either the word containing the current file position or the last sequence of constituent characters before the file position.

`forward-word` `[Function]`

Arguments: *pane*

Package: `text-edit`

- moves the file position to the end of the next word. **`forward-word`** does nothing if the file position is at the end of *pane*.

`backward-word` `[Function]`

Arguments: *pane*

Package: `text-edit`

- moves the file position to the start of the previous word. **`backward-word`** does nothing if the file position is at the start of *pane*.

`kill-word` `[Function]`

Arguments: *pane*

Package: `text-edit`

- cuts the next word out of *pane* and pushes it onto the Lisp Clipboard.

`backward-kill-word` `[Function]`

Arguments: *pane*

Package: `text-edit`

■ cuts the previous word out of *pane* and pushes it onto the Lisp Clipboard.

delete-word [Function]

Arguments: *pane*

Package: text-edit

■ deletes the next word from *pane* without copying it to the Lisp Clipboard.

backward-delete-word [Function]

Arguments: *pane*

Package: text-edit

■ deletes the previous word from *pane* without copying it to the Lisp Clipboard.

upcase-word [Function]

Arguments: *pane*

Package: text-edit

■ converts all alphabetic characters in the current word to upper case.

downcase-word [Function]

Arguments: *pane*

Package: text-edit

■ converts all alphabetic characters in the current word to lower case.

capitalize-word [Function]

Arguments: *pane*

Package: text-edit

■ Converts the first of each sequence of alphanumeric characters in the current word to upper case. The table below shows some examples of the effect of **capitalize-word.**

Before	After
select	Select
SELECT	Select

Before	After
SELECT-CURRENT-WORD	Select-Current-Word
select2words	Select2words
one+one	One+One

`select-current-word` [Function]

Arguments: *pane*

Package: `text-edit`

- selects the current word (see Section 3.9, **Operations on Regions**).

`current-symbol` [Function]

Arguments: *pane*

Package: `text-edit`

- selects the current word and returns a symbol whose print name is that word.

3.16.3 Operations on lines

`beginning-of-line` [Function]

Arguments: *pane*

Package: `text-edit`

- moves the file position to the start of the current line.

`end-of-line` [Function]

Arguments: *pane*

Package: `text-edit`

- moves the file position to the end of the current line.

`kill-line` [Function]

Arguments: *pane*

Package: `text-edit`

- cuts text from the current position to the end of the line, leaving the #\Newline character, and pushes the text onto the Lisp Clipboard. If the file position is at the end of a line, a newline character #\Newline is cut instead.

`backward-kill-line` [Function]

Arguments: *pane*

Package: `text-edit`

- cuts text from the current position to the start of the line, leaving the #\Newline character, and pushes the text onto the Lisp Clipboard. If the file position is at the start of a line, a newline character #\Newline is cut instead.

`delete-line` [Function]

Arguments: *pane*

Package: `text-edit`

- deletes text (excluding #\Newline) from the current position to the end of the line. If the file position is at the end of a line, the newline character #\Newline is deleted instead.

`backward-delete-line` [Function]

Arguments: *pane*

Package: `text-edit`

- deletes text (excluding #\Newline) from the current position to the start of the line. If the file position is at the start of a line, the newline character #\Newline is deleted instead.

`next-line` [Function]

Arguments: *pane*

Package: `text-edit`

- moves the current file position down one line. The insertion point is kept in the same column if possible. The file position is moved to the end of the destination line if the line is too short to allow indentation to be preserved.

`previous-line` [Function]

Arguments: *pane*

Package: `text-edit`

■ moves the current file position up one line. The insertion point is kept in the same column if possible. The file position is moved to the end of the destination line if the line is too short to allow indentation to be preserved.

`newline` [Function]

Arguments: *pane*

Package: `text-edit`

■ inserts a newline character #\Newline. This breaks the current line at the cursor position and leaves the file position at the start of the new line after the #\Newline character.

`open-line` [Function]

Arguments: *pane*

Package: `text-edit`

■ inserts a newline character and then leaves the file position just before the line break.

3.16.4 Operations on Lisp forms

The definitions of "next" and "previous" Lisp forms are similar to those used for words. The Text Editor ignores any comments when performing operations on forms and treats them as whitespace.

The *current* or *next* form is identified by the context around the current file position.

- If the file position is within a symbol, that symbol is the current form.
- Otherwise, the current form is the next form which would be read by the reader. There is therefore no current form if the file position is immediately before a closing parenthesis.

The *previous* form is found as follows:

- If the file position is within or just after a symbol, the previous form is that symbol.
- Otherwise it is the form which would be found by reading backwards from the file position. In the example below,

```
#'(lambda (x y) (cons x y))
```

with the file position after the final closing parenthesis, the previous form is the whole expression *excluding* the characters #'.

- If the file position is in the middle of a symbol, `delete-sexp` etc. delete characters from the symbol to the right or left of the file position, as appropriate.

`forward-sexp` [Function]

Arguments: *pane*

Package: `text-edit`

■ places the file position after the end of the current form. Beeps if there is no current form.

`backward-sexp` [Function]

Arguments: *pane*

Package: `text-edit`

■ places the file position before the start of the previous form. Beeps if there is no previous form.

`kill-sexp` [Function]

Arguments: *pane*

Package: `text-edit`

■ cuts text between the current file position and the end of the current form and pushes it onto the Lisp Clipboard.

`backward-kill-sexp` [Function]

Arguments: *pane*

Package: `text-edit`

■ cuts text between the current file position and the start of the previous form and pushes it onto the Lisp Clipboard.

`delete-sexp` [Function]

Arguments: *pane*

Package: `text-edit`

■ deletes text between the current file position and the end of the current form.

backward-delete-sexp [Function]

Arguments: *pane*

Package: text-edit

■ deletes text between the current file position and the start of the previous form.

3.16.5 Operations on lists

The Text Editor provides extensive facilities for manipulating Lisp programs as lists. The Editor makes some assumptions about the structure of the program text which enable it to identify enclosing lists when moving through a file

If the first character in a line is an open parenthesis, then this is treated as the beginning of a form.

insert-empty-list [Function]

Arguments: *pane*

Package: text-edit

■ inserts the empty list ( ) at the file position and places the insertion point between the parentheses.

forward-list [Function]

Arguments: *pane*

Package: text-edit

■ moves to the end of the next list at the current depth of nesting. If there are no more lists at the current depth, **forward-list** beeps and leaves the file position unchanged.

backward-list [Function]

Arguments: *pane*

Package: text-edit

■ moves to the start of the previous list at the current depth of nesting. If there are no more lists at the current depth, **backward-list** beeps and leaves the file position unchanged.

forward-up-list [Function]

Arguments: *pane*

Package: text-edit

■ places the file position immediately after the closing parenthesis of the enclosing list. If there is no enclosing list, **forward-up-list** beeps and leaves the file position unchanged.

backward-up-list [Function]

Arguments: *pane*

Package: text-edit

■ places the file position immediately before the opening parenthesis of the enclosing list. If there is no enclosing list, **backward-up-list** beeps and leaves the file position unchanged.

down-list [Function]

Arguments: *pane*

Package: text-edit

■ moves the file position to immediately after the opening parenthesis of the next list at the current level. If there are no more lists at the current level, **down-list** beeps and leaves the file position unchanged.

3.16.6 Operations on definitions

The Text Editor considers a *definition* to be a list whose opening parenthesis is in the left-most column.

beginning-of-definition [Function]

Arguments: *pane*

Package: text-edit

■ moves the file position backwards to the start of the preceding definition. If there is no preceding definition, **beginning-of-definition** leaves the file position at the start of *pane*.

beginning-of-next-definition [Function]

Arguments: *pane*

Package: text-edit

■ moves the file position forwards to the start of the next definition. If there are no more definitions in the file, **beginning-of-definition** leaves the file position at the end of *pane*.

end-of-definition [Function]

Arguments: *pane*

Package: text-edit

■ moves to the end of the current definition. If the file position is already at the end of a definition, **end-of-definition** moves to the end of the next definition in the file if there is one. If there are no more definitions, **end-of-definition** beeps and leaves the file position at the end of *pane*.

eval-definition [Function]

Arguments: *pane*

Package: text-edit

■ moves the file position backwards to the start of the preceding definition then reads it using the function **read** and evaluates it using *top-eval*. Any results of the evaluation are printed to the Toploop *pane*.

3.16.7 Operations on comments

find-start-of-comment [Function]

Arguments: *pane*

Package: text-edit

■ places the file position immediately before any semicolon introducing a comment on the current line. If there is no comment, the file position is set to the end of the current line.

kill-comment [Function]

Arguments: *pane*

Package: text-edit

■ cuts comment text from the current line, including all preceding semicolons, and pushes it onto the Lisp Clipboard. If there is no comment on the line, **kill-comment** does nothing.

indent-for-comment [Function]

Arguments: *pane*

Package: text-edit

■ moves the insertion point to the column given by the value of *comment-indent* and inserts a single semicolon and a space. If the tab position is already beyond *comment-indent*, a new line is inserted and the comment is placed on the new line.

3.16.8 Indentation

The following indentation functions estimate the required whitespace based on an analysis of the current lisp definition. They work with both fixed and variable width fonts.

reindent-single-line [Function]

Arguments: *pane*

Package: text-edit

■ indents the current line relative to the current list by adding an appropriate amount of whitespace. The file position of *pane* should be at the start of a line when **reindent-single-line** is called. After the call it is moved to the end of any inserted whitespace. This function does nothing if the previous line is all whitespace or there are net closing super-brackets on it.

reindent-sexp [Function]

Arguments: *pane*

Package: text-edit

■ reindents all lines starting within the form immediately following the current file position.

newline-and-indent [Function]

Arguments: *pane*

Package: text-edit

■ inserts a new line at the current file position and calls **reindent-single-line** on the new line.

comment-newline-and-indent [Function]

Arguments: *pane*

Package: text-edit

■ inserts a new line at the current file position and calls **reindent-single-line**. In addition, if the line starts with a comment, any semicolons and whitespace after them are duplicated on the new line.

delete-horizontal-space [Function]

Arguments: *pane*

Package: text-edit

■ deletes continuous whitespace to the left and right of the current position. The operation stops at the first non- whitespace or #\Newline character in either direction.

delete-indentation [Function]

Arguments: *pane*

Package: text-edit

■ removes any indentation on the current line by deleting any whitespace characters at its start. The file position is left at the start of the line.

3.16.9 Operations on regions

Every Text Edit pane has a currently selected region which is displayed inverted or otherwise highlighted. In addition to the functions below, the generic functions **copy-selection**, **paste-selection**, and **delete-selection** may be used to manipulate selected regions (see the Edit menu Items entry in the Common Graphics section of the Online Manual).

set-region [Function]

Arguments: *pane start end*

Package: text-edit

■ selects the region between the file positions *start* and *end*. The function has no effect if *end* is before *start*.

select-all [Function]

Arguments: *pane*

Package: text-edit

■ equivalent to **set-region** on the entire contents of *pane*.

get-region [Function]

Arguments: *pane*

Package: text-edit

■ returns two integer values giving the start and the end of the selected region. Both these values are equal to the current file position if no region is selected.

read-region [Function]

Arguments: *pane*

Package: text-edit

■ returns the characters between the start and end of the selected region as a Lisp string.

pretty-print-region [Function]

Arguments: *pane*

Package: text-edit

■ reads all objects which start in the selected region and pretty-prints them back to *pane*, overwriting the original text.

reindent-region [Function]

Arguments: *pane*

Package: text-edit

■ reindents all lines which start in the selected region. Sets the file position to the end of the selection.

delete-to-kill-buffer [Function]

Arguments: *pane*

Package: `text-edit`

- cuts the selected text from *pane* and pushes it onto the Lisp Clipboard.

`copy-to-kill-buffer` [Function]

Arguments: *pane*

Package: `text-edit`

- copies the selected text from *pane* and pushes it onto the Lisp Clipboard.

`yank-from-kill-buffer` [Function]

Arguments: *pane*

Package: `text-edit`

- inserts text from the top of the Lisp Clipboard at the current file position and leaves the file position at the end of the selection.

3.16.10 Operations on panes

`scroll-one-line-up` [Function]

Arguments: *pane*

Package: `text-edit`

- scrolls *pane* one line upwards. Does nothing if *pane* is at end of file.

`scroll-one-line-down` [Function]

Arguments: *pane*

Package: `text-edit`

- scrolls *pane* one line downwards. Does nothing if *pane* is at start of file.

`number-of-lines-in-window` [Function]

Arguments: *pane*

Package: `text-edit`

- returns the number of lines of text in the visible portion of *pane*.

`next-page` [Function]

Arguments: *pane*

Package: `text-edit`

- scrolls *pane* upwards by one pane full of text minus `*number-of-lines-kept-in-page-scroll*`.

`previous-page` [Function]

Arguments: *pane*

Package: `text-edit`

- scrolls *pane* downwards by one pane full of text minus `*number-of-lines-kept-in-page-scroll*`.

3.16.11 Operations on files

`beginning-of-file` [Function]

Arguments: *pane*

Package: `text-edit`

- moves the file position to the start of *pane*.

`end-of-file` [Function]

Arguments: *pane*

Package: `text-edit`

- moves the file position to the end of *pane*.

3.16.12 Mark operations

Marks are used to remember file positions for later reference. A marked place moves with the text around it to reflect any deletions or insertions made between the file start and the mark. If the text around a mark is deleted, the mark moves to the place previously occupied by the start of the deleted region.

`make-mark` [Function]

Arguments: *pane position*

Package: `text-edit`

- creates a mark in *pane* at the specified file *position*.

mark-p [Function]

Arguments: *object*

Package: text-edit

- non-nil if *object* is a mark.

mark-position [Function]

Arguments: *mark*

Package: text-edit

- returns the current location of *mark*.

delete-mark [Function]

Arguments: *pane mark*

Package: text-edit

- removes the link between the *mark* and the text in the *pane* allowing more rapid updating of *pane*.

Every pane automatically has a unique principal mark. The following functions act specifically on this mark.

:mark [Generic Function]

Arguments: *pane*

Package: keyword

- sets the principal mark in *pane* to the current file position.

get-mark [Function]

Arguments: *pane*

Package: text-edit

- returns the position of *pane*'s principal mark. If it has not yet been set, **get-mark** returns the current file position.

:select-to-mark [Generic Function]

Arguments: *pane*

Package: keyword

■ selects the region between the current file position and the principal mark.

:exchange-to-mark [Generic Function]

Arguments: *pane*

Package: keyword

■ moves the current file position to the principal mark. The mark is then moved to the file position just abandoned. Nothing is done if the principal mark has not yet been set.

3.16.13 Textual search and replace

string-search [Function]

Arguments: *pane target* &optional *backward case-sensitive*

Package: text-edit

■ searches for the string target in *pane*. The search proceeds forward from the current file position unless *backward* is non-nil. Upper and lower case characters are considered identical unless *case-sensitive* is non-nil. Returns t if the string was found.

string-replace [Function]

Arguments: *pane target replacement-text*
&optional *backward case-sensitive globally*

Package: text-edit

■ searches for the string target in *pane* and replaces it with *replacement-text*. If *globally* is non-nil all occurrences of *target* are replaced. The search proceeds forward from the current file position unless *backward* is non-nil. Upper and lower case characters are considered identical unless *case-sensitive* is non-nil. Returns t if replacement was performed.

:find [Generic Function]

Arguments: *pane*

Package: keyword

■ displays a dialog which requests a search string and options and then calls **`string-search`**.

`:find-same` [Generic Function]

Arguments: *pane*

Package: `keyword`

■ looks for another occurrence of the last search string using the same search options. If the text is not found or there is no current search string, **`:find-same`** beeps and does nothing.

`:find-clipboard` [Generic Function]

Arguments: *pane*

Package: `keyword`

■ uses the top item of the Lisp Clipboard as the target search string. If it is not a string, a dialog is displayed which allows the user to convert the object or abandon the search.

`:replace` [Generic Function]

Arguments: *pane*

Package: `keyword`

■ displays a dialog which prompts the user for a target string, a replacement string and search options and then carries out the replacement operation.

`:replace-same` [Generic Function]

Arguments: *pane*

Package: `keyword`

■ carries out a string replacement operation using the last specified target string, replacement string and options.

3.16.14 Access to information and documentation

The following functions provide information on the next symbol in the source text. If the file position is inside a symbol, that symbol is used; otherwise, information is printed on the symbol found by searching forwards through the text.

:apropos [Generic Function]

Arguments: *pane*

Package: keyword

■ searches for all symbols which have the print name of the current symbol as part of their own print name, and for each symbol prints information to *help-output* about its definition and dynamic binding. **apropos** is described in the Online Manual.

:lambda-list [Generic Function]

Arguments: *pane*

Package: keyword

■ prints the lambda list of the current symbol to the stream that is the value of *help-output*.

:documentation [Function]

Arguments: *pane*

Package: keyword

■ prints the documentation associated with the current symbol to *help-output*. See the entry on **documentation** in the Online Manual.

:describe [Function]

Arguments: *pane*

Package: keyword

■ prints information about the current symbol to *help-output* by calling the function **describe** (see the entry on **describe** in the Online Manual).

:build-call [Generic Function]

Arguments: *pane*

Package: keyword

■ takes the current symbol as a function, analyses its lambda list and pushes a prototype call onto the top of the Lisp Clipboard. **:build-call** does nothing if there is no function definition associated with the current symbol.

:find-definition [Generic Function]

Arguments: *pane*
Package: keyword

- attempts to find source code for the current symbol.

3.16.15 Access to debugging tools

The functions described in this section give access from text editors to the trace, breakpoint and profile facilities of Allegro CL for Windows. These debugging tools are further described in Chapter 6 of this manual. The symbol passed to the appropriate debugging macro must have an associated global function definition.

:trace [Generic Function]

Arguments: *pane*
Package: keyword

- calls the macro **trace** with the current symbol as its argument.

:breakpoint [Generic Function]

Arguments: *pane*
Package: keyword

- calls **breakpoint** with the current symbol as its argument.

:profile [Generic Function]

Arguments: *pane*
Package: keyword

- calls **profile** with the current symbol as its argument.

:untrace [Generic Function]

Arguments: *pane*
Package: keyword

- **untrace**s the current symbol.

:unbreakpoint [Generic Function]

Arguments: *pane*

Package: keyword

- **unbreakpoint**s the current symbol.

:unprofile [Generic Function]

Arguments: *pane*

Package: keyword

- **unprofile**s the current symbol.

3.16.16 Recursive editing

:inspect [Generic Function]

Arguments: *pane*

Package: keyword

- invokes the inspector according to a decision made in a dialog to edit all forms in the selected region. **:inspect** does nothing if there are no lists within the selection.

3.16.17 Miscellaneous

:undo [Generic Function]

Arguments: *pane*

Package: keyword

- undoes the effect of the most recent operation in *pane*. A second call to **:undo** redoes it.

:revert-to-saved [Generic Function]

Arguments: *pane*

Package: keyword

- discards any changes made to the contents of *pane* since it was last saved.

:modified-p [Generic Function]

Arguments: *pane*

Package: keyword

- non-nil if changes have been made to *pane* but have not been saved.

clear-modified-flag [Function]

Arguments: *pane*

Package: text-edit

- clears the modified flag of *pane*.

insert-empty-string [Function]

Arguments: *pane*

Package: text-edit

- inserts two " (double quote) characters and leaves the file position between them.

brackets-matched-p [Function]

Arguments: *pane start end*

Package: text-edit

- returns a non-nil value if between *start* and *end* all opening parentheses are balanced by at least the same number of closing parentheses.

count-bracket-mismatch-between-positions [Function]

Arguments: *pane start end*

Package: text-edit

- counts the parenthesis mismatch between file positions *start* and *end*. Ordinary parentheses and super-brackets are dealt with together. The parenthesis mismatch between the two positions is returned. A positive mismatch value means that there are too many opening parentheses.

3.16.18 Loading and saving files

edit-file [Function]

Arguments: *dummy* &optional *pathname*

Package: text-edit

■ opens a file for text editing. If *pathname* is nil or not supplied, **cg:pop-up-open-file-dialog** is used to obtain a *pathname*. If the file is already being edited, the appropriate Text Edit pane is selected. If not, a new pane is opened and **load-file** is used to copy the text into it.

load-file [Generic Function]

Arguments: *pane* &optional *file*

Package: text-edit

■ copies the text from file into *pane*. *file* should be a namestring or pathname referring to an existing file which can be opened using :element-type character or string-char. If *file* is nil or is not supplied, the function **cg:pop-up-open-file-dialog** is used to obtain a pathname. After **load-file** has been called, (**file** *pane*) returns *file*. **load-file** returns t to indicate success, or nil if the user aborted the loading process.

analyse-definitions-on-opening [Variable]

Package: text-edit

■ if non-nil text files will be scanned to record the definitions in them, whenever they are opened. If nil this analysis is suppressed.

file [Generic Function]

Arguments: *pane*

Package: text-edit

■ returns the pathname of the file in which the contents of *pane* are saved or nil if it has not yet been saved. The value returned by **file** is affected by **load-file** and **save-file.**

save-file [Function]

Arguments: *pane* &optional *file*

Package: text-edit

■ copies the text in *pane* to *file*. *file* must be a namestring or pathname. If *file* is not supplied or its value is nil, **cg:pop-up-open-file-dialog** is used to obtain a pathname. *file* is created if it does not already exist. If it does exist, it is deleted and a new file with the same name is created. After **save-file** has been called, (**file** *pane*) returns *file*, and (**modified-p** *pane*) returns nil. **save-file** returns t to indicate success, or nil if the user aborted the saving process.

:save [Generic Function]

Arguments: *pane*

Package: keyword

■ equivalent to (**save-file** *pane* (**file** *pane*)).

If the contents of a text-edit-window need to be saved, the title will show an asterisk (*). This asterisk is not part of the title string. This behavior is controlled by the following variable.

flag-modified-text-windows-p [Variable]

Package: text-edit

■ When this variable is true, the titles of text-edit-windows that need to be saved will contain an asterisk (*). The asterisk is not part of the title string. When this variable is nil, no such asterisk is displayed. The initial value is t.

3.16.19 Symbol completion and lambda lists

quick-lambda-list [Function]

Arguments: *window* &optional (*always-p* t)

Package: text-edit

■ This function displays lambda-list information in the status bar about the current symbol of *window* (that is, the symbol at the text cursor in window). If the symbol has a function binding, then the type of function binding (function, generic function, macro, or special form) is displayed at the left, followed by the symbol name, followed by the names of the function's parameters.

If the current symbol has no function binding but does have a variable binding, then the value of the variable is displayed. Other messages are displayed if the current symbol has no binding, or if the string at the text cursor does not name a symbol.

If the symbol name string contains a package prefix, then that package is searched for the symbol, otherwise the package of window is searched.

If *always-p* is non-`nil`, then the message is always displayed as described above. If it is `nil`, then the message is printed only when there is a bound symbol, the lambda list is displayed only if the symbol name string is immediately preceded by either an open parenthesis or a single quote, and no message is displayed if the previous status bar message was a quick-lambda-list for that same symbol.

quick-lambda-list-and-insert-space [Function]

Arguments: *window* &optional *always-p*

Package: text-edit

■ Like **te:quick-lambda-list**, but also inserts a space at the text cursor in window, so that this function can handily be assigned to the space bar as it is by default in the built-in comtabs. Note that the default value of *always-p* is `nil` (its default is `t` for **te:quick-lambda-list**).

complete-symbol [Function]

Arguments: *window*

Package: text-edit

■ This function enters characters into *window* so as to complete a partially-typed symbol name. If there is only one possible completion, then the characters are inserted immediately; otherwise a menu is popped up containing the possible completions. If a choice is selected from the pop-up menu, then characters are inserted so as to complete the selected symbol; otherwise no completion is done. A symbol can be selected from the pop-up menu either by clicking it or by typing the single letter displayed to the left of each symbol on the menu.

If there are more than `te:*max-symbol-completion-choices*` (defined next) possible completions, then a message indicating this is printed in the status bar, no menu is popped up, and no completion is done.

If the symbol name string contains a package prefix, then that package is searched for the symbol, otherwise the package of window is searched.

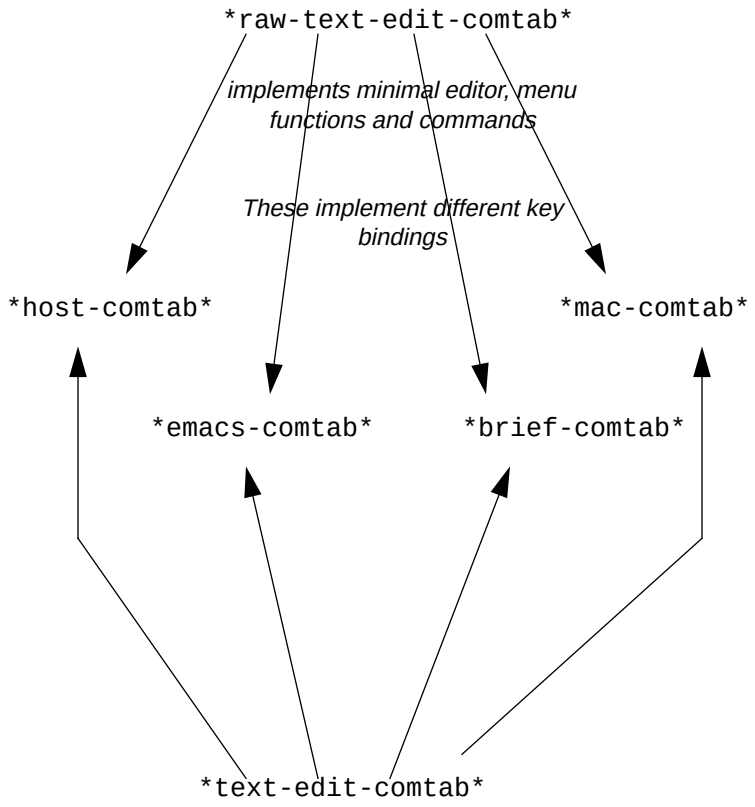
`*max-symbol-completion-choices*` `[Variable]`

Package: `text-edit`

■ The maximum number of symbol completions that will be displayed in a symbol completion pop-up window. If more than this number of possibilities exist, that fact will be printed to the status bar and no symbol completion window will be displayed. the initial value of this variable is 50.

3.16.20 Text editor comtabs

The diagram below illustrates the relationship between the Text Editor comtabs. For more general information on comtabs, refer to Chapter 9 of this Guide.


raw-text-edit-comtab
implements minimal editor, menu functions and commands
These implement different key bindings
host-comtab
mac-comtab
emacs-comtab
brief-comtab
text-edit-comtab
inherits from the keybinding comtabs according to the text edit mode

[This page intentionally left blank.]

Chapter 4 The Inspector

The Inspector takes its name from its ability to inspect, and in most cases, edit the contents of any Lisp data structure. The Inspector can make dynamic changes to bitmaps, array elements, structures and other objects that have already been defined in your Lisp system. Since the changes are effected through a standard interface, the Inspector is easy to use. In conjunction with the Debugger, it is particularly suited to tracking down problems and patching up erroneous values.

Bringing up an inspector window 1: right-button menu

Depressing the right mouse button over a selected object displays a menu of choices appropriate to the object. For most Lisp objects, the top item in the right button menu is the name of the object itself. Choosing that top item causes an inspector window for the object to be displayed.

Bringing up an inspector window 2: Tool/Inspect menu

You can also select a selected object by choosing the menu item **Inspect Selected Object** from the Inspect submenu of the Tools menu. (Ctrl-I is the keyboard equivalent).

Bringing up an inspector window 3: inspect function

`inspect` takes a Lisp object as its one argument and displays an inspector window for that object. Thus `(inspect 'foo)` display an inspector window for the symbol `foo`.

Closing inspector windows

Inspectors windows can be closed by choosing **Close** from the File menu (or the Manage submenu of the Window menu) or by clicking in the close box of the window. Choosing **Close All Inspectors** from the Inspector submenu of the Tools menu closes all inspector windows.

Inspecting fixnums

Fixnums are such uninteresting objects that a call to inspect one is taken as a call to change its value, so a type-in window asking for a new value is displayed rather than an actual inspector window.

4.1 Using the Inspector - an example

The best introduction comes from guiding you through a simple example which edits the value of a slot in a Lisp structure. The Lisp source code you need can be found part way through the file called *\ex\lang\09data.lsp*, supplied with your Lisp system. Open the file and scroll through it until you find the region with `**Defining structures**` shown below. You can of course simply type in and evaluate the expressions rather than work from the file supplied on disk. We have opened the file, found the location of interest and selected the structure definition by double-clicking on the opening parenthesis:

Press Alt-Enter, which causes the structure definition to be evaluated, creating a new type of Lisp object called `person`. The result of the evaluation is printed in the Toploop window.

The slot names in the structure are `familyname`, `sex`, `age` and `occupation`, all of which default to the symbol `unknown`.

Now, in the 09data.lsp window, go down to the second **`setf`** form (starting `(setf ;Defining person fred with known components.`), scrolling if necessary. Double-click on the parenthesis preceding that **`setf`**.

then press Alt-Enter. The **`setf`** form is evaluated and the result is printed in the Toploop window.

You have created a `person` structure representing Mr. Smith, a thirty year-old programmer, held as the value of the symbol `fred`. Consider what would be necessary to modify, say, the age of `fred` from the numerical value of 30 to the text "Thirty". In Lisp, the code would be:

```
(setf (person-age fred) "Thirty")
```

In this expression, **`person-age`** is known as an accessor function since it accesses the `age` slot of a `person` structure (represented here by the symbol `fred`) and returns the value found there. This is all very well if you are familiar with the many accessor function names of Common Lisp, but the advantage of the Inspector lies in providing a more intuitive way of achieving the same result without a detailed knowledge of Lisp.

Try making the change to `fred` outlined above by using the Inspector. The first step is to select the text `fred` anywhere on the screen. We select it in the 09data.lsp window:

Choose **Inspect Selected Object** from the Inspect submenu of the Tools menu. Many items in menus have keystroke equivalents. These are given on the right-hand end of the line containing the item in the menu. Thus, for example, Ctrl-I is the keyboard shortcut for **Inspect**. However it is invoked, the result is an Inspector window appears, displaying the value of each of the symbol `fred`'s cells on a separate line.

The Inspector works recursively and can be used to probe further into any of the data structures shown, but only values preceded by a * (`Value`, `Function`, and `plist` in the illustration) can be modified. In nearly all instances, changes made to objects are to the originals, i.e. they are destructive. If you make a mistake or want to backtrack over your steps, you can repeatedly select **Undo** from the Edit menu. If you wish to go back several steps, **Undo to Before** may be more convenient. **Revert to Saved** in the File menu takes you back to the first entry of the Undo history.

Note that one line (the first in the illustration above) is shaded, indicated it is selected. You can select a line by clicking over the line or by moving up with the ↑ key or down with the ↓. Select the line of text displaying the value of `fred` by using either the cursor keys ↑↓ or by clicking somewhere on that line.

We want to further inspect the value. To do so, you can, as before, choose **Inspect Selected Object** from the Inspect submenu of the Tools menu; or use the keyboard equivalent Ctrl-I. Further, within the Inspector and Debugger Backtrace windows, you can also double-click on an item. Perform any one of these actions and a new Inspector window opens up showing the slots within a Lisp `person` structure.

Most Lisp objects use this screen layout for the Inspector window. The first line specifies the object under inspection; each subsequent line carries the name of a slot followed by its value. The slot names depend on the object. Values preceded by a * can be modified.

We want to modify the value of the `age` slot. To modfiy a slot (that can be modified), simply select it by clicking over it and type a space (or any key). A type in area appears over the value of the field, as shown in the next illustration. (We have already typed the new value, the string "Thirty".)

Type in the string “Thirty”, as shown,and press Enter. The text should be enclosed in quotes so that it is read into Lisp as a string, not as a symbol name. If you decide not to change the value after you have started typing, press the Escape key and the type-in area will disappear with the value unchanged.

After Enter is pressed. the value is changed. You can see this be evaluating `(person-age fred)` in the Toploop window (we have cleared the earlier contents of Toploop):

You can still undo the changes by choosing **Undo** from the Edit menu. If you were to exit the Inspector now (by closing the Inspector windows), the modified value of “Thirty” would be saved as a permanent change to `fred`.

To continue with the example, double-click on the `age` slot to inspect the string “Thirty”.

"Thirty" is inspected as a string in terms of its constituent characters which you can edit just as before. The point to note is that the Inspector has built-in methods for displaying each type of Lisp object. You will see later how you can inspect a Lisp bitmap in a pictorial representation rather than as text.

Returning to the example, close the last window and click on the `occupation` slot of the structure.

We want to modify the value from `programmer` to `analyst`. Again, with the occupation line selected (as in the illustration above), simply start typing. A type-in area appears when you press the first keyboard key (but this character is not entered). Then type the new value. In the illustration above, we have already typed *ANALY*.

Finish typing *ANALYST*. As soon as you press Enter, Lisp updates the value (press Escape to cancel the change before pressing Enter):

If you entered a string, the update occurs as soon as the closing `"` is entered. Again, you can undo the change by choosing **Undo** from the Edit menu.

In this example the symbol has been automatically uppercased. Such effects are controlled by the reader and printer preferences (see **Setting Your Preferred Lisp Environment** in Chapter 2 of this manual).

The Undo facility has already been mentioned. The Edit menu items **Cut**, **Copy**, **Paste**; the **Backspace** and **Delete** keys are all available with the restriction that non-`*`'ed items can only be copied. **Cut** is the combination of **Copy** followed by **Delete**.

To illustrate this, select the `familyname` slot containing the symbol `smith` then choose **Copy**. The value is copied into the Lisp Clipboard. Select the `occupation` slot followed by **Paste**. The new symbol is inserted. The next illustration shows the Inspector window after this operation:

The Delete and Backspace keys and **Delete** in the Edit menu attempt to set the slot to a default value. In the case of `fred`, the structure was defined with slots defaulting to the symbol `unknown`. This can be verified by clearing one of the slots.

Select the `familyname` slot containing the symbol `smith` then press Backspace. The structure slot is cleared to its default value `unknown`.

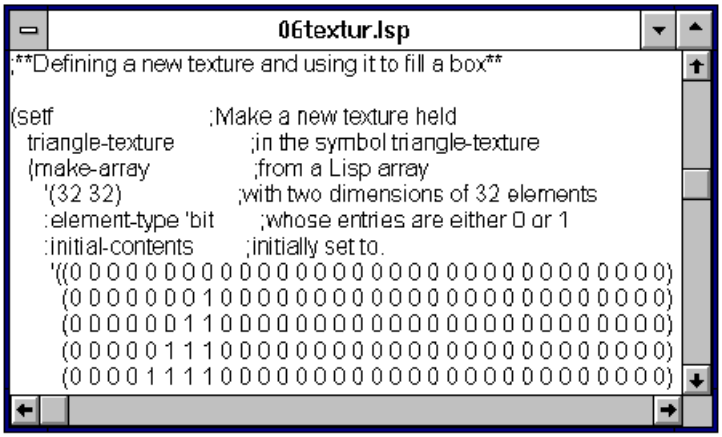
To leave the Inspector, choose **Close All Inspectors** from the Inspect submenu of the Tools menu. Or, after first making sure an Inspector window is uppermost, select **Close All** from the File menu. All changes to the inspected object are automatically saved.

4.2 Inspecting Bitmaps

A feature of the Inspector is the way in which it can be configured to display different types of Lisp object in different ways. For most objects, a text window suffices with each line carrying information about a slot. An exception is the Bitmap Editor which can be used instead of a textual editor whenever a bitmap object or bitmap definition is inspected. This section guides you through an example. The Lisp source code you need can be found part way through the file *\ex\cg\misc\06textur.lsp* supplied with your Lisp system.

Open the file and scroll through it until you find the region shown below. You can type in and evaluate the expressions rather than work from the file supplied.

> **Warning: unlike edits made with the standard Inspector window, bitmap edits cannot be undone, so take care to make copies of important bitmaps and edit the copies.**

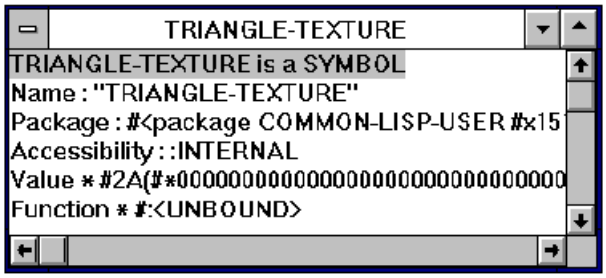


Double-click on the bracket preceding **`setf`**, so the whole form is selected. Then press Alt-Enter. The form will be evaluated and the result printed in the Toploop window.

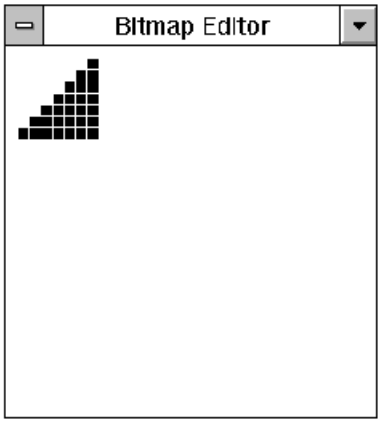
We have defined a Lisp texture represented by the symbol `triangle-texture`. A texture is a type of bitmap: a two-dimensional array whose elements are 0 or 1. Their use

is described in the *Common Graphics Introductory Guide*, but they are introduced here to demonstrate the Bitmap Editor.

Select the symbol `triangle-texture` then select **Inspect Selected Object** in the Inspect submenu of the Tools menu to inspect its value.



The cell values of the symbol `triangle-texture` are displayed. Double-click on the `value` slot.



A fixed-size Bitmap Editor window appears. By default, the Inspector uses the Bitmap Editor whenever the inspected object is of the appropriate type. The contents of the bitmap are mapped onto the screen as a grid. A black square signifies a 1 in the bitmap at that point; a white square, a 0.

Use the mouse to edit the bitmap. Clicking on a square inverts its state, so white 0 goes to black 1 and black 1 goes to white 0. If you depress the mouse then drag to a new position, all squares under its path will be set to the same state as at the starting position. As with many things, it is simpler to do than say, so try editing the `triangle-texture` bitmap.

Experiment with clicking and dragging the mouse on the bitmap.

When you are satisfied with your revised bitmap, close the Bitmap Editor window. Note again that the bitmap is destructively modified as you edit so there is no way to retrieve the original bitmap except by re-evaluating the code that produced the bitmap in the first place. For this reason, we recommend editing copies of bitmaps rather than originals. When you close the Bitmap Editor window, the underlying Inspector window is updated to reflect the changes made.

Conclude the demonstration by closing any remaining Inspector windows.

4.3 Inspecting System Data

Inspecting the current status of system variables is particularly simple. A list of useful values is available on a menu to make access easy. The information can be helpful during debugging. Choose **Inspect System Data** from the Inspect submenu of the Tools menu.

The sub-submenu lists useful system objects. Since some of the objects are currently in use, you are only allowed to inspect a copy of the original. For example, it is not wise to destructively modify a texture since that would modify all uses of the texture at once.

Select the required object from the entries, or click outside the menu to make no selection.

4.4 Inspector Preferences

You can alter some of the default actions of the Inspector. For example, it is possible to inspect bitmaps as Lisp arrays rather than by using the Bitmap Editor.

Choose **Main Preferences** from the Preferences menu and click on the Inspector tab. The following form appears:

Most entries are self-explanatory. The two variables defined just below, `*inspect-all-slots*` and `*sort-inspected-slots*`, allow users who preferred the 2.0 behavior to the new 3.0 behavior to get it (as the variable definitions describe). `*inspect-length*` limits the total number of slots displayed in any one Inspector window. The boxes next the boolean variables indicate whether the value is true or false. A check in the box means the value is true, an empty box means the value is false. You can modify the variables as desired with the mouse. Your changes will be effected when you click on **Apply** (masked in the illustration but unmasked as soon as any value is changed) or **Save** (which also does an apply as well as saving the preferences to a file). Clicking on **Close** discards your changes and leaves the variables unmodified while **Revert** undoes all modifications since the were last Applied or Saved (or original values if neither **Apply** or **Save** have been clicked on). Clicking **Close** closes the window.

Clicking on either of the font button brings up a dialog allowing you to set the font used in Inspector windows.

Variables that preserve release 2.0 inspector look-and-feel

There are two changes in release 3.0 compared to release 2.0: all slots of an object are displayed (rather than just some) and slots are listed alphabetically. This behavior is controlled by the following two variables, which are both initially true. Set them to `nil` if you preferred the 2.0 behavior.

`*inspect-all-slots*` [Variable]

Package: `inspector`

■ If non-`nil`, then the built-in inspect-object methods that otherwise display only some of the slots of a standard-object will instead display them all. Initial value is `t`.

`*sort-inspected-slots*` [Variable]

Package: `inspector`

■ If non-`nil`, then the default inspect-object methods for standard-object and structure-object will list the slots alphanumerically. Initial value is `t`.

4.5 Using The Inspector - Summary

Here is a summary of the steps required to use the Inspector:

1. **Select a Lisp object**. In most cases, do this by double-clicking in the object's symbol but you can also select a printed representation of an object displayed in an Inspector window.
2. **Open the Inspector**. Choose **Inspect Selected Object** from the Inspect submenu of the Tools menu. Ctrl-I is a quicker keystroke alternative. Within the Inspector and Debugger windows you can also double-click on an item. A window opens displaying the slots of the inspected object.
3. **Inspect/Edit the attributes of the object as required**. Inspect an attribute further by double-clicking on it. Edit by pasting in a new value, pressing backspace to clear the existing value (replacing it with its default if any is defined), or type in Lisp text directly from the keyboard. Changes can usually be undone, step by step.

4. **Close the Inspector**. Choose **Close All Inspectors** from the Inspect submenu of the Tools menu. Or make sure an Inspector window is active and select **Close All** from the File menu. All changes are automatically saved.

4.6 Inspector internals

This section was chapter 4 of the *Inside Progamming Tools* manual in release 2.0.

The window-based Inspector is used to examine Lisp data structures of any complexity. It allows you to browse recursively through objects and make changes to **setf**able fields.

Earlier sections in this chapter provide an introduction to using the Inspector on your system. This chapter describes the functions which allow you to invoke the Inspector under program control, define new inspectors or modify the actions of existing ones.

The Inspector consists of two parts: the *Inspector Pane* with its associated comtab, and the *Inspection Manager*. The Inspector Pane handles screen events via its command table. The Inspection Manager defines the way that objects are displayed, including their slot names and the way in which the slots may be updated. It also installs any extra comtab entries and menus required by the Inspector.

Typically the Inspector displays the contents of an object as slot-value pairs. A slot which has an associated **setf** method may be modified (cleared, cut, pasted etc.). Otherwise the slot is read-only, and operations are restricted to recursive inspecting, copying and so on.

Most symbols naming functions, variables, etc. discussed in this chapter are in the `inspector` package. This package is not used by default by `common-lisp-user` and is not in the default use list of the various package creation functions. Therefore, you must either use the `inspector` package or qualify the symbols (with `inspector:` or `insp:`) when referring to them.

4.6.1 Program interface

The programmers interface to the Inspector is provided by the following functions.

inspect [Function]

Arguments: *object* &key (:windup-fn #'identity)
Package: common-lisp

■ invokes the Inspector on *object* using a method which depends on the type of *object*. :windup-fn is a function of one argument which is called on exit from **inspect** and by the **Save** and **Save as...** menu options. It is responsible for saving *object* back to its place of origin. The default, #'identity, does not attempt to save *object*.

edit-bitmap [Function]

Arguments: *bitmap windup-fn*

Package: inspector

■ calls the bitmap editor to edit *bitmap*, which must be a two-dimensional bit array. *windup-fn* is a function of one argument (the edited *bitmap*) which is called on exit from **edit-bitmap** and should save the *bitmap* to its place of origin.

4.6.2 Inspector control variables

The special variables described in this section allow you to control the way in which many Lisp types are inspected.

inspect-length [Variable]

Package: inspector

■ is the maximum number of slots displayed in an Inspector Window. The initial value of *inspect-length* is 300.

inspect-list [Variable]

Package: inspector

■ controls the inspection of lists. *inspect-list* may have the following values:

- :sequence. Where possible, lists are inspected as sequences. If the list is circular or the last cons in the list is a dotted pair, the Inspector may invoke the Structure Editor.
- :plist. The Inspector displays lists as property-value pairs. To be inspected as a plist, the list must have an even number of members, and the first and every odd item must be symbols.

- `:alist`. The inspector displays lists as association lists. Every member of a list must be a cons for it to be displayed in this form.

■ The value of `*inspect-list*` does not compel the inspector to display lists in the requested format. If the inspected object is incompatible with `*inspect-list*`, it is inspected using the Structure Editor. The Structure Editor is always used, regardless of the value of `*inspect-list*`, if the list is circular.

`*inspect-bit-vector-as-sequence*` [Variable]

Package: `inspector`

■ if this variable is non-`nil`, bit vectors are inspected as sequences. Otherwise, they are inspected in their usual printed representation (`#*...`).

`*inspect-bitmap-as-array*` [Variable]

Package: `inspector`

■ if non-`nil`, bitmaps (two-dimensional bit arrays) are inspected as arrays; otherwise, the bitmap editor is used.

`*inspect-string-as-sequence*` [Variable]

Package: `inspector`

■ if this variable is non-`nil`, strings are inspected as sequences. Otherwise, they are inspected in their usual printed representation ("...").

`*inspect-structure-as-sequence*` [Variable]

Package: `inspector`

■ if this variable is `nil`, structures are inspected as slot name-value pairs. Otherwise, structures are inspected simply as numbered slots, as if the structure were a vector. This overrides any specific methods for types which are implemented as structures. For example, if building is defined as a structure:

```
(defstruct building (height ...
```

and `*inspect-structure-as-sequence*` is non-`nil`, buildings will be inspected as sequences irrespective of any inspection method which may have been defined for the type building.

`*sequence-structure-slots-settable*` [Variable]

Package: `inspector`

■ if non-`nil`, then the slots in any structures displayed as sequences are made settable. You should exercise caution in using this option, because slots which have been declared `:read-only` may be modified.

4.6.3 Defining new inspectors

The Allegro CL for Windows Inspector is totally customizable. The **`inspect-object`** generic function allows you to inspect your own data types or system data types in any format you wish. For most Lisp objects, the slot-value pair format provided by the functions **`inspect-with`** and **`inspect-with-slots`** is appropriate, but for some objects (such as bitmaps) a radically different approach is more useful.

It is important to remember that the Inspector is divided into two parts. The Inspection Manager decides which object slots are to be displayed and how the slots are updated. It also adds any extra command table entries and menus required, but it does not actually handle any events. These are the responsibility of the Inspector Pane, which takes care of screen handling and command dispatch. The interface between the two parts is in terms of a non- negative integer called the slot index which identifies each field within a Lisp object. There is usually a simple relationship between the slot index, the corresponding field in the Lisp object and the position of the slot in the Inspector window. In the case of a vector, slot index n could have a simple interpretation as `(aref` *`object`* `n)`, and the corresponding element would be displayed on line n+1 of the Inspector window. However, it would be possible for the Inspector to display the elements in reverse order, or perhaps to omit some of them, by changing the mapping between the index and vector element.

For example, if a cons is inspected in the usual line-by-line style:

a description of the cons is printed on line 0 of the window;
slot index 0 is its car (displayed on line 1); and
slot index 1 is its cdr (on line 2 of the window).

`inspect-object` `[Generic Function]`

Arguments: *`(object type) windup-fn`*
Package: `inspector`

■ defines an Inspector for objects of *type*, which may be any user-defined structure type or one of the following system types:

array	bignum	bit-vector	bitmap
character	complex	cons	fixnum
float	list	ratio	string
structure	symbol	vector	

■ The first argument is the object to be inspected. The second is a windup function which is responsible for saving the object when it has been inspected. See **inspect-with** for a description of the windup function.

■ It is not usually necessary to write completely new Inspectors, since **inspect-with** and **inspect-with-slots** provide most of the facilities needed. In the example below, an Inspector for complex numbers is defined using **inspect-with-slots** to carry out almost all of the processing.

```
(in-package 'inspector)
(defmethod inspect-object ((number complex) windup-fn)
    (inspect-with-slots number windup-fn #'true
        '(("Real part" realpart)
           ("Imaginary part" imagpart))))
```

inspect-with [Function]

Arguments: *object windup-fn open-fn slot-name-fn slot-value-fn* &optional *(slot-validp-fn* #'false*) slot-setting-fn slot-default-fn*

Package: inspector

■ inspects *object* using the standard slot-value format. The action of the Inspector is controlled by the functions passed as parameters to **inspect-with**.

■ *windup-fn* is a function of one parameter which may be called when the Inspector for *object* is closed. The parameter is the modified version of *object*. *windup-fn* is responsible for saving it to wherever is appropriate. If the original and inspected forms of *object* are **eql**, *windup-fn* is not called at all. Pass *#'false* if you do not need to save *object*.

■ *open-fn* initializes the Inspection Manager. *open-fn* is a function of two parameters:

```
(defun specific-open-fn (window object) (...
```

where the first argument is the inspector window and the second is the object to be inspected. Typically you would use this function to install specific menus and command table entries.

■ *slot-name-fn* is a function of two arguments: the object being inspected and a slot index. This function should return the name of the slot with the given index, or `nil` if the index is out of range. The index is guaranteed to be non-negative.

■ *slot-value-fn* is a function of two arguments: the object being inspected and a slot index. *slot-value-fn* should return the value of the slot with this index.

■ *slot-validp-fn* is used to check values before installing them in *object* slots. It is a function of three arguments: the object, slot index and value. This function should return a non-`nil` value if the value can be installed at the given slot index. The default value, `#'false`, always returns `nil` and allows none of the slots to be changed.

■ *slot-setting-fn* is a function of three arguments: the object, slot index and value. *slot-setting-fn* should set the appropriate slot to the specified value. This function is only called if *slot-validp-fn* has indicated that the value is acceptable.

■ *slot-default-fn* is a function of two arguments: the object and a slot index. *slot-default-fn* should return a form which can be evaluated to produce the default value of the specified slot. If this function is absent, the slots cannot be cut or deleted.

inspect-with-slots [Function]

Arguments: *object windup-fn open-fn slot-descriptions*

Package: `inspector`

■ provides a declarative way of entering the Inspector. The *object, windup-fn* and *open-fn* arguments are identical to the corresponding arguments of **inspect-with**.

■ *slot-descriptions* is a list containing a description of each slot in *object*. Each slot description is a list containing:

- The slot name as a string or symbol.
- The accessor function fn such that (fn *object*) returns the value of the slot.
- A default value to be installed if the slot is cut or deleted (see **inspect-with).**
- **The** type of *object* held in the slot. This is used to vet any new values stored in this field. Use `t` if the *object* can hold data of any type.

Supply only the first two slot description items if the slot cannot be modified.

The example below shows a declarative inspector for the building type. The location, use and value fields are displayed; the location is read-only, and the value field can store integers. Cut or Delete operations set value to zero.

```
(defmethod inspect-object ((object building) windup-fn)
    (inspect-with-slots object windup #'true
                        '(("Location is" building-location)
                          ("Use is" building-use :unknown t)
                          ("Value is" building-value 0 integer))))
```

4.6.4 Inspector panes

inspected-object [Function]

Arguments: *pane*

Package: `inspector`

■ returns the object being inspected in *pane*, which must be an `inspector-pane` or an `inspector-window`.

redisplay-inspector-window [Function]

Arguments: *pane*

Package: `inspector`

■ makes the Inspector redisplay *`pane`*, which must be a `structure-edit-pane`. This function should be called when an inspected object has been changed by side effects (for example, by the action of a pop-up menu).

4.6.5 An example

This section gives a complete example of an inspector written using both **`inspect-with-slots`** and **`inspect-with`**. It shows how to write all the support functions which may be needed by the Inspector, but it is far longer and more detailed than most Inspectors you will need to write. Most Inspectors need no more than a simple declarative call to **`inspect-with-slots`**, with no associated *`open-fn`*.

The example installs an Inspector for the type point, which is a user-defined structure with three fields: an identity (x, vertex-1 etc.), the point's x-coordinate and its y- coordinate. point-x and point-y are stored in Cartesian coordinates, lending itself naturally to an implementation using **`inspect-with-slots`**. However, we also wish to be able to inspect points in circular coordinates. The method used is controlled by the value of `*inspect-points-as-circular*`. In this case, the inspection manager needs to map between the slot values and the displayed representation using the transformation formulae

$x = r\cos\theta$ $\quad y = r\sin\theta$

$r = (x^2+y^2)$ $\quad q = \tan^{-1}(y/x)$

The implementation of inspection in circular coordinates uses **`inspect-with`**, and this example shows you how to define the auxiliary functions needed to provide access to point slots.

First, the initialization of variables and the definition of point. The *`id`* field is made `:read-only`.

```
(in-package 'inspector)

(defstruct point
    (id nil :read-only t)
        (x 1 :type number)
        (y 1 :type number))
```

```
(setq a (make-point :id 'vertex-1 :x 2 :y 3))
(setq b (make-point :id 'vertex-2 :x -3 :y 6.2))
(defvar *inspect-point-as-circular* nil)
```

Now, the definition of the Inspector itself. The Inspector body simply calls **`inspect-point`**. This in turn looks at the value of `*inspect-point-as-circular*`. If it is `nil`, **`inspect-with-slots`** is called. Otherwise, **`inspect-with`** is called to inspect the point in circular coordinates.

```
(defmethod inspect-object ((object point)windup-fn)
    (inspect-point point windup-fn))

;;; The main inspection function for points. If
;;; *inspect-point-as-circular* is false, points are
;;; inspected in Cartesian format. This is done by
;;; inspect-with-slots, which just extracts the
;;; values from the appropriate structure fields.
;;; Only x and y slots are settable.
;;;
;;; Inspection in circular form involves translation
;;; between the internal x-y representation and
;;; r-theta form. It uses inspect-with together with
;;; appropriate support functions.

(defun inspect-point (point windup-fn)
    (if *inspect-point-as-circular*
        (inspect-with
            point
            windup-fn
            #'inspect-point-open-fn
            #'inspect-circular-slot-name-fn
            #'inspect-circular-value-fn
            #'inspect-circular-validp-fn
            #'inspect-circular-setting-fn
            #'inspect-circular-default-fn)
        (inspect-with-slots
            point
            windup-fn
            #'inspect-point-open-fn
```

```
            '(("Identity" point-id)
            ("x-coord" point-x 0 number)
            ("y-coord" point-y 0 number)))))
```

The *open-fn* adds a pop-up menu to the Inspector pane when it is opened. This contains four items: (1,0), (0,1) (1,0) and (0, 1). Choosing one of them sets the point to the chosen value. The Inspector pane then has to be redisplayed so that it shows the correct value.

```
;;; Definition of the Inspector open-fn.
;;; inspect-point-open-fn attaches the pop-up menu to
;;; the point inspector pane.

(defun inspect-point-open-fn (pane object)
(set-window-menu pane 'menu *inspect-point-menu*))

;;; inspect-point-menu returns a menu which can then be
;;; attached to the Inspector window. It contains four
;;; items. The value returned by a menu selection is of
;;; the form
;;;
;;; '(function-to-call x-value y-value)
;;;
;;; This is then used by the menu selection function
;;; do-inspector-menu-command

(defun inspect-point-menu ()
    (setq *inspect-point-menu*
        (open-menu
            '(#S(menu-item :name "(1,0)"
                :value (set-point 1 0))
            #S(menu-item :name "(0,1)"
                :value (set-point 0 1))
            #S(menu-item :name "(-1,0)"
                :value (set-point -1 0))
            #S(menu-item :name "(0,-1)"
                :value (set-point 0 -1)))
            'pop-up-menu *lisp-main-window* :selection-function
            #'do-inspector-menu-command)))
```

```
;;; For convenience, set a special variable to the
;;; menu.

(defvar *inspect-point-menu* (inspect-point-menu))

;;; do-inspector-menu-command takes apart the structure
;;; returned by the pop-up menu and calls the appropriate
;;; function. It includes the window and inspected
;;; object in the argument list.

(defun do-inspector-menu-command (menu menu-item stream)
    (let ((window (selected-window (selected-window-stream)))
          (menu-item-value (menu-item-value menu-item)))
      (funcall
          (first menu-item-value)     ; The function to call
          window                      ; Inspector window
          (inspected-object window)   ; The point.
          (second menu-item-value)    ; X coordinate to set.
          (third menu-item-value))))  ; Y coordinate to set.

;;; set-point changes the inspected point in response to
;;; a pop-up menu selection.
;;; It changes point-x and point-y, then tells the
;;; Inspector to redisplay the window because the
;;; Inspector doesn't know about the change.

(defun set-point (window point x y)
    (setf
        (point-x point) x
        (point-y point) y)
    (redisplay-inspector-window window))
```

The functions defined above support the Cartesian inspection mode completely, including its pop-up menu. The remainder of the code provides the support functions needed by **`inspect-with`** to handle the circular format.

```
;;; Functions for inspecting points in circular format.
;;; They do not check that the values installed
;;; are non-complex.
```

```
;;; The slot name function returns “Identity”, “R”,
;;; “Theta” and nil. nil indicates that the last slot
;;; has been reached. This is the only function which
;;; has to be able to handle out-of-range values of the
;;; slot index.

(defun inspect-circular-slot-name-fn (point index)
    (aref '#("Identity" "R" "Theta" nil) index))

;;; The slot value function returns the ID field
;;; unchanged, but converts the internal Cartesian
;;; representation to R-Theta form.
;;; The value function simply returns the value of
;;; the first slot, but calculates the "R" and "Theta"
;;; slot values from the x and y coordinates.

(defun inspect-circular-value-fn (point index)
    (case index
        (0 (point-id point))
        (1 (point-r point))
        (2 (point-theta point))))

;;; The validp function checks values as follows:
;;;
;;; Slot 0 (id)              No new values allowed (slot is
;;;                          read-only).
;;; Slot 1 (R)               Positive numbers allowed.
;;; Slot 2 (Theta)           Any number allowed (but the
;;;                          value shown in the Inspector
;;;                          window is always in the range
;;;                          -pi to pi).

(defun inspect-circular-validp-fn (point index value)
    (and
        (not (zerop index))
        (numberp value)
        (or
            (= index 2)
```

```
            (plusp value))))

;;; The slot setting function. This function is never
;;; called with index = 0 because the ID slot cannot be
;;; set. For the “R” and “Theta” slots, it transforms
;;; back to Cartesian coordinates and updates the x and
;;; y slots. Note that the Inspector calls the
;;; slot-value function when it displays the new values,
;;; so the values of theta shown in the Inspector window
;;; are always in the range (-pi, pi).

(defun inspect-circular-setting-fn (point index value)
    (if (= index 1)           ; Never called with index = 0
        (let ((theta (point-theta point)))
            (setf
                (point-x point) (* value (cos theta))
                (point-y point) (* value (sin theta))))
        (let ((r (point-r point)))
            (setf (point-x point)
                (* r (cos value))
                (point-y point) (* r (sin value))))))

;;; The default value for the theta slot is zero, and
;;; for R is 1 (R = 0 is meaningless).

(defun inspect-circular-default-fn (point index)
    (if (= index 1) 1 0))

;;; Two support functions. point-r and point-theta
;;; carry out the Cartesian-to-circular transformation.

(defun point-r (point)
    (sqrt
        (+
            (square (point-x point))
            (square (point-y point)))))

(defun point-theta (point)
    (atan (point-y point) (point-x point)))
```

Finally, a call which inspects a `point` object.

```
(inspect a)
```

Now try a circular inspection:

```
(setf *inspect-point-as-circular* t)
(inspect b)
```

4.6.6 Default window sizes

The default sizes of windows associated with the inspector are controlled by the following variables and the functions **`cg:default-window-width`** and **`cg:default-window-height`**. These methods, specialized for inspector windows, will return the value of the `*inspector-window-width*` or `*inspector-window-height*` (respectively) if that value is non-`nil`. If the value is `nil` (as it is initially for both variables), then the default methods will return an integer equal to the `*inspector-window-width/height-factor*` value multiplied by the interior size of the `*lisp-main-window*`.

`*inspector-window-width*` [Variable]

Package: `inspector`

- The initial value is `nil`.

`*inspector-window-height*` [Variable]

Package: `inspector`

- The initial value is `nil`.

`*inspector-window-width-factor*` [Variable]

Package: `inspector`

- The initial value is 0.6.

`*inspector-window-height-factor*` [Variable]

Package: `inspector`

- The initial value is 0.4.

[This page intentionally left blank.]

Chapter 5 Trace, breakpoint and profile

No matter how careful a programmer tries to be, the first draft of a new section of code frequently contains bugs. The compiler will detect some errors as the code is read in and these can easily be corrected and the code recompiled. Other errors, however, will not manifest themselves until the code is executed. This is particularly true in languages such as Lisp where the type of an argument is not usually known until runtime. For instance, the code:

```
(defun foo (x) (car x))
```

is perfectly correct syntactically, as is:

```
(defun bar () (foo 1))
```

and both will be compiled without complaint, but on typing:

```
(bar)
```

a runtime error is signaled because **car** is called with a number as its argument rather than a list.

Incremental testing of new functions (i.e. interactively testing new functions using sample data soon after they are defined) often serves to isolate the bug to a few lines of code where subsequent inspection by eye can locate the problem. Unfortunately not all bugs can be found in this way and when a program fails mysteriously deep within a computation, you may have little idea of what caused the error.

There are various steps which you can take when faced with this situation. Some programmers will prefer to use the Debugger (see Chapter 6) to investigate the state of the system when the error is signaled. This can be rather a sledgehammer approach but is very effective when the source of the problem is believed to be close to the function which signaled the error, or when you are desperate for some hint to follow up. It has the drawback

that it may not display the actual cause of the error, since the erroneous function may no longer be executing. Moreover, the Debugger is of limited use if the bug causes an incorrect value to be returned rather than signaling an error. Simple inspection of the source code is usually helpful but may not be sufficient as it provides no information about the flow of control which led to the error. If this flow of control can be examined, it will often be easy to spot at which point the program started to behave in an unexpected fashion. One way of getting such information is to insert calls to **`print`** or **`debugger`** into the code and recompile it. Another way is to use the Trace facility.

5.1 Simple Tracing

To get to grips with the trace package, start up Lisp and define the factorial function **`fact`** as follows:

```
(defun fact (n) (if (<= n 1) 1 (* n (fact (1- n)))))
```

For the non-mathematical, the factorial of a positive whole number is obtained by multiplying together all the whole numbers between it and 1. Thus, factorial of 5 = 5 x 4 x 3 x 2 x 1 = 120. Our definition of **`fact`** therefore expects a number as its argument. It is best to give it a small positive number for this demonstration, but it can handle larger numbers happily.

Check that `(fact 1)` returns 1 and `(fact 5)` returns 120. Try `(fact 500)` if you wish.

The Trace dialog

Output from tracing can be voluminous. In the default, it goes to the Toploop window but if the Trace dialog (illustrated below) is displayed, output goes there. You display the trace dialog by choosing **Trace Dialog** from the Trace submenu of the Tools menu.

Starting tracing

Now, type:

```
(trace fact)
```

and then:

```
(fact 5)
```

The Trace dialog window should look something like this:

Most of this output has come from **trace** and is generated at the time the traced function is called. The trace information should be fairly self-explanatory in this simple case: the calls are shown in a call graph (not very interesting in this simple recursive function). Note that deep recursion may cause the graph to go beyond the edge of its area of the dialog. We have selected one call (its highlighted) and the argument and returned value of that call are displayed, as is the stack (in the lower right). Unless you clear the dialog (by clicking on the **Clear** button), the next call to **fact** will be combined with the call reported here.

Counter values. There are several other counters maintained by trace, breakpoint and profile which are described later in this chapter, but it is worth remembering that redefining a traced (or breakpointed or profiled) function will cause all the counters to be reset to zero. Try redefining **fact** and calling it again to see this. Notice also that **defun** knows that **fact** is traced and keeps it traced.

Trace output. Trace writes to the Trace dialog if it is available. Otherwise, the output or **trace** goes to the stream `*trace-output*`, which is initially a synonym stream of `*terminal-io*`.

Controlling the printer. The special variables `*trace-print-level*` and `*trace-print-length*` are used to control the printer while **trace** is printing. These variables appear on the Printer form of the Preferences dialog, displayed by choosing **Main Preferences** from the preferences menu.

Infinite recursion. You should be careful if you trace certain system functions. Tracing those used by the Printer or by **format** could potentially cause infinite

recursive calls to be performed when Lisp tried to print the trace information. In fact, Lisp avoids this by setting a flag to prevent the recursion, but consequently not all calls to the traced function will actually be traced. If you think you may have entered an infinite loop, try to break out by pressing the Break key (sometimes labeled 'Break Pause').

Removing tracing

You can remove tracing from **`fact`** by typing:

```
(untrace fact)
```

or simply:

```
(untrace)
```

which untraces everything currently traced.

5.2 Conditional Tracing

If you suspect that a function is failing only occasionally, say for a particular set of arguments, you can cause trace information to be printed only when given those arguments. Likewise, you may just want to see what the function returns in a particular situation. **`trace`** allows a conditional test to be performed on each call of a traced function and will only print trace information for that particular call if the result of the test is not `nil`.

The conditional test may be any Lisp expression but to help you Lisp makes three values available to the expression by binding local variables. The current call number is bound to the variable `call-count`, the current call depth to `call-depth` and a list of the current arguments to `args`. Try experimenting with this by defining **`fact`** and then typing:

```
(trace (fact (= call-depth 3)))
```

Be careful to get the parentheses right. Then:

```
(fact 5)
```

causes the trace information to be printed only when exactly three calls of **`fact`** are in progress. Executing:

```
(fact 5)
```

a second time produces similar output, but the call count has changed. Now enter:

```
(trace (fact (= call-count 2)))
```

Retracing a traced function causes the tracing counters for that function to be reset to zero. In particular, the call counter is now zero, so:

```
(fact 5)
```

causes information to be printed for the second call, but executing:

```
(fact 5)
```

again displays no information as the call counter runs from 6 to 10 during the calculation, so the test always fails.

Printing additional information. You can put calls to **`print`** and **`format`** in the conditional expressions to get more information out.

Default conditionals. `(trace fact)` is precisely equivalent to `(trace (fact t))`; that is, tracing **`fact`** with a conditional which always succeeds.

5.3 Tracing Lots of Things

You may have wondered why a set of parentheses was required around **`fact`** and its conditional test when the test was specified. This is because a single call to **`trace`** can trace more than one thing. If you have three functions called **`foo`**, **`bar`** and **`baz`** each of which takes two arguments, then you can trace them all, with various conditionals, by typing:

```
(trace (foo (= call-count 29))
       bar
       (baz (and (= call-depth 3)
                 (eq (first args) 'hello)))
)
```

`foo` and **`baz`** have conditionals and every call to **`bar`** is traced. The same effect can be achieved more verbosely by:

```
(trace (foo (= call-count 29)))
(trace bar)
(trace (baz (and (= call-depth 3)
                 (eq (first args) 'hello))))
```

You can find out which functions are currently traced just by typing `(trace)`. If you have been following the examples so far, Lisp will respond:

```
(foo bar baz)
```

5.4 Tracing Places

What you have seen so far is fine for tracing functions defined by **`defun`**, or installed by **`set-symbol-function`**.

> **Installing new symbol functions**. You should not use **`set-symbol-function`** on a symbol whose function definition is being traced, because only **`defun`** knows about tracing.

Suppose you want to trace calls to an object which is stored somewhere else, such as in an array or on the property list of a symbol. Functions are often stored in such places to allow data-driven programming. As a rather artificial example, suppose that in part of a program you want to call a function on arguments *`arg1`* and *`arg2`* depending on the value of *`arg0`*, which you expect to be an integer in the range 0 to n. You have defined an array called `func-array` containing (n+1) elements, each of which is the function we want to call for that value of *`arg0`*. Quite reasonably, you call the appropriate function by typing:

```
(funcall (aref func-array arg0) arg1 arg2)
```

You then discover problems when you call the function in slot 5 of the array, and decide to trace it. **`trace`** is no use, but **`tracef`**[1] comes to the rescue.

```
(tracef (aref func-array 5))
```

does the job. **`tracef`** causes calls to the code stored in any place acceptable to **`setf`** to be traced. It takes a conditional, which defaults to `t`, but unlike **`trace`**, it can only trace one thing at a time and so the parentheses around the traced object and its conditional can be omitted. This restriction is to avoid ambiguity over what should be traced. Thus, you would write:

```
(tracef (aref func-array 5) (= call-depth 1))
```

1. Note that tracef is an extension to Common Lisp. The symbol tracef is in the allegro package.

to trace top-level calls to the code stored in:

```
(aref func-array 5)
```

You might like to try defining **fact**, then:

```
(setq func-array
      (make-array 1 :element-type t :initial-element 'fact)
(tracef (aref func-array 0))
(funcall (aref func-array 0) 5)
```

Untracing places. Calling **untracef** on a place acceptable to **setf** untraces the place if it has been traced. You can use `(untrace)` to untrace absolutely everything, both places and functions.

5.5 Setting Breakpoints

Tracing provides information about a call to a code object and about the values returned by the call, but the information and control it provides are by no means complete. It does not, for instance, show where the code object was called from, and it does not give you the option of aborting the calculation if you see that it is certain to fail because the arguments are wrong or a function is returning the wrong values. A way is needed of interrupting the execution of the program just before and just after execution of the code object in question. This can be done using breakpoints.

A breakpoint is a point where the flow of execution of a program can be interrupted, or broken. The variables and data structures of the program can be examined and perhaps modified before resuming execution of the program. You can try setting breakpoints in **fact**. First type `(untrace)` to untrace everything from the previous examples, then define **fact** and type:

```
(breakpoint (fact (= call-depth 3))) (fact 5)
```

Lisp displays a Restarts dialog when the breakpoint is reached:

Click **Invoke Selected Restart** while the `return from break` restart is selected and shortly you will see a second Restarts dialog:

Invoke the `return from break` restart again and the result 120 will be printed by the Toploop as usual.

The first dialog box was displayed just before starting the third recursive call to **`fact`**, while the second was displayed just after the result of that call had been obtained. At either time, clicking **Enter Debugger** would enter the Lisp Debugger (and clicking **Abort** would abort the **fact** computation altogether). **`breakpoint`** is very like **`trace`** in that many functions can be breakpointed using a single call to **`breakpoint`**, with or without conditional tests involving their call count, call depth and arguments, using the same syntax as **`trace`**. **`unbreakpoint`** is precisely analogous to **`untrace`**, and **`breakpointf`** and

unbreakpointf are provided and work like **tracef** and **untracef** respectively. **breakpoint** keeps its own counters separate from those used by **trace**.

With breakpoints still set on **fact**, type:

```
(trace (fact (= call-depth 3)))
(fact 5)
```

You can see that a code object can be traced and breakpointed at the same time and, for a given call, trace information is displayed before a chance to break is offered. This is so that the arguments and results can be examined before deciding on a course of action.

5.6 Profiling

Gathering information about the runtime characteristics of a program is called profiling. You can accumulate timing information about the execution of code objects and display this information in a tabular format. Such information can show where a program is spending most of its time and hence which areas of the program could most usefully be speeded up. To illustrate the data which profiling can provide, define the following function:

```
(defun foo ()
  (for i from 1 to 60000 do
     (+ 1 2)))
```

This is just a loop doing a little work in the middle.

> **The for loop**. If you are not familiar with the **for** loop, see the description of the **for** macro in the Online Manual.

The Profile dialog

In the default, profiling output goes to the Toploop window but if the Profile dialog (illustrated below) is displayed, output goes there. You display the trace dialog by choosing **Profile Dialog** from the Profile submenu of the Tools menu.

Starting profiling

You can use the **`profile`** macro to examine the performance of **`foo`**. The syntax for **`profile`** is just like **`trace`** and **`breakpoint`**, so type:

```
(profile foo)
```

and then:

```
(foo)
```

When the call completes, the Profile dialog will contain information on the time spent, number of calls, and supply a call graph.

The graph on the left shows the calls (all called, profiled functions are shown). **foo** is somewhat uninteresting because its call structure is simple, so let us look at a slightly more complicated example.

Recall **fact** defined in the trace section above and also used in the breakpoint example. Make sure it is untraced and unbreakpointed:

```
(untrace fact)
(unbreakpoint fact)
```

Now define **bar**, **baz**, and **buz**. **Baz** and **buz** simply call **fact** but because **fact** is so recursive, if it is profiled, it will clutter up the graph.

```
(defun baz (n) (fact n))
(defun buz (n) (fact n)
(defun bar (n)
  (let ((a (baz n))
        (b (buz (* n 2)))
(print (/ b a))))
```

Profile all three functions:

```
(profile bar baz buz)
```

and do

```
(bar 22)
```

The profile dialog looks like:

One thing that may be a surprise is that **buz** and **baz** take a small portion of the total

time. In fact, printing a large number (which work is done by **bar**) takes much longer than calculating it.

In the graph, the open folder icons represent intermediate calls (they call something else) while the page-with-folded-corner icons represent leaves -- that is functions that do not call other (profiled) functions. You can trim the graph by clicking on the minus (-) box at the nodes of the graph or lengthen it by clicking on the plus (+) box at a terminal node (unless the terminal node is a leaf). Further calls to profiled functions accumulate on the graph until the **Clear** button is clicked, which clears the profile record and the dialog.

Getting results without the Profile dialog

If the Profile dialog is not displayed, profile results (without the call graph and with less data) can be displayed with the macro profile-results or by choosing **Profile Results** from the profile submenu of the Tools menu. The same call to **foo** produces the following output on the Toploop window:

```
Function  Profile count  Total time  Average Max  Min
FOO       1              3.08        3.08    3.08 3.08
```

- `Function` names the function being profiles (**foo** in this case).
- `Profile count` is the number of times that the conditional test associated with profiling **foo** returned a non-`nil` result. In the example, the test was omitted and so defaulted to `t`, which means that the profile count is the same as the number of times **foo** has been called since last being profiled or redefined.
- `Total time` displays the total time spent in **foo** while the timer was running. Every time **foo** is called when the condition was non-`nil`, a timer is started and runs until **foo** returns from the call. The sum of these times (in seconds) is displayed in the `Total time`.
- `Average` is the `Total time` divided the `Profile count` (since `Profile count` is 1, `Average` is the same as `Total Time`.
- `Max` is the time of the longest run. Since there was only one run, it is the same as `Total time`.
- Min is the time of the shortest run. Since there was only one run, it is the same as `Total time`.

Try running **foo** a few more times and examining the results again. You may see some slight variation in the maximum and minimum times. This is due to the fact that the timer

counts in distinct ‘ticks’, rather than running continuously, so results are rounded to the nearest tick. Also, other activity within the computer, such as interrupts occurring, can increase the execution time.

> **Garbage collection**. The time taken for garbage collection is not subtracted from the execution time of a function in which it occurs, so you should be careful of this when using the profiler.

Profiling overheads

A further warning must be given: there is some overhead associated with running the profiler. Enter:

```
(unprofile)
```

just in case something is still profiled, then:

```
(defun foo ()
  (for i from 1 to 30000 do
     (bar)))
(defun bar ()
  (for i from 1 to 2 do
     (+ 1 2)))
(profile foo)
(foo)
(profile-results)
```

`(profile-results)` on its own prints accumulated information about all profiled functions and places (in case you had not guessed by now, you can profile and unprofile places using **`profilef`** and **`unprofilef`** in exactly the same way as tracing them). The Toploop window should look like this:

Now type:

```
(profile-reset foo)
```

to clear the profile information about **foo**, then:

```
(profile bar)
(foo)
(profile-results)
```

Again, it may be hard to read the information in the illustrations. They say that calling **foo** once while **bar** is not being profiled takes about 1.04 seconds while calling it while **bar** is also profiled takes about 10.11 seconds![1].

Profiling the right thing

If you define **fact** as before and then enter:

```
(profile fact)
(fact 5)
```

what you will have timed is:

```
<the time for (fact 5)> +
<the time for (fact 4)> +
<the time for (fact 3)> +
<the time for (fact 2)> +
<the time for (fact 1)>
```

when what you probably wanted was:

```
<the time for (fact 5)>
```

because a new timer is started for each profiled call to **fact**. You can restrict the profiling by using:

```
(profile (fact (= call-depth 1)))
```

which only times the outermost call.

5.7 Menu Commands

The **Trace**, **Breakpoint**, and **Profile** are all choices on the Tools menu. Each displays an associated submenu which provides a convenient way of enabling or disabling the tracing, breakpointing or profiling of a selected symbol without a conditional. Using this command can save a lot of unnecessary typing.

1. Note that timings depend on machine configuration, loading, and processor clockrate. You may get very different actual times. However, you will see an increase in the time it takes foo to be executed when bar is profiled regardless of your machine configuration.

To use any of the submenus, simply select the symbol and then the operation to be performed on it. If you select something other than a symbol, the first symbol within the selection will be used.

The **Status** entries print information to the status bar about what is being traced, breakpointed, or profiled.

Many of the menu items can be chosen with buttons on the toolbar. The Break items show a hammer, the trace items a zig-zag, and the profile items (not on the toolbar by default) show a human profile. To see exactly what button does what, move the mouse cursor over the button and look in the status bar.

5.8 Trace, breakpoint, and profiling internals

This section was chapter 6 of the *Inside Programming Tools* manual in release 2.0.

5.8.1 Tracing functions and places

Calls to a function can be traced, telling the user how many calls have been made to the function, the depth of recursion and the arguments with which the function was called. Additionally, Allegro CL for Windows also allows the user to trace functions in **setf**able places.

`*trace-print-level*` [Variable]

Package: `allegro`

■ controls the maximum depth to which arguments and results of traced functions are printed by the trace subsystem. Its initial value is 4.

`*trace-print-length*` [Variable]

Package: `allegro`

■ controls the maximum length to which arguments and results of traced functions are printed by the trace subsystem. Its initial value is 10.

`trace` [Macro]

Arguments: *`{function-description}*`*

Package: `common-lisp`

■ traces the functions and macros provided. If no *`function-descriptions`* are provided, all traced functions and places are displayed.

☞ In Common- Lisp, **`trace`**'s arguments may only be symbols which name the functions or macros to be traced. In Allegro CL for Windows each *`function-description`* is either a symbol or a list of a symbol and a test-form, which may be any Lisp expression. The test-form is evaluated each time the traced function or macro is called, and determines whether trace information should be displayed for that particular call. During execution of the form, the following variables are bound appropriately:

- `call-count` - the number of calls to this function since it was made traceable.
- `call-depth` - the current depth of recursive calls to the function.
- `args` - list of the arguments of the particular call.

For example, to cause trace information to be displayed for function **`foo`** when it is called for the 17th time,

```
(trace (foo (= call-count 17)))
```

could be used. `test-form` may be an arbitrary expression, and permits tracing to be very specific. In the following example, **`foo`**'s first argument is printed when `*my-count*` is greater than 100, and all other **`trace`** output is suppressed by returning `nil`.

```
(trace (foo
       (if (> *my-count* 1000)
           (progn (print (first (args))) nil)))))
```

■ **`trace, breakpoint`** and **`profile`** count calls separately.

`tracef` [Macro]

Arguments: *`place [test-form]`*

Package: `allegro`

■ traces calls to the function stored in *`place`*. Trace information will only be displayed if *`test-form`* returns a non-`nil` value. *`test-form`* defaults to `t`.

`untrace` [Macro]

Arguments: *`{function}*`*

Package: `common-lisp`

■ untraces function. If none is given, all traced functions and places are untraced.

☞ In standard Common Lisp, **`untrace`** only affects traced functions.

`untracef` [Macro]

Arguments: *`place`*

Package: `allegro`

■ untraces the function stored in *place*. This macro has no effect if *place* is not already traced. *place* should be untraced before a new function is stored into it.

5.8.2 Setting breakpoints

Breakpoints can be introduced to a section of code so that the program stops running when these points are reached. This is helpful if, for some reason, the code is not being executed properly, and the user wishes to check whether or not individual sections are running as they should. If a section of code is terminated with a breakpoint and the code is executed correctly up to this point, then the user knows that this section is not causing the problem.

breakpoint [Macro]

Arguments: *{function-description}**

Package: allegro

■ sets breakpoints on the given functions, causing a call to **break** before and after the given function call. *function-description* behaves in the same way as for **trace**.

breakpointf [Macro]

Arguments: *place [test-form]*

Package: allegro

■ sets a breakpoint at *place*, causing a call to **break** before and after a call to the function stored at *place*.

unbreakpoint [Macro]

Arguments: *{function}**

Package: allegro

■ removes the breakpoints from the functions given. If none are given, all breakpointed functions and places are unbroken.

unbreakpointf [Macro]

Arguments: *place*

Package: allegro

■ removes the breakpoint from *place*. This should be done before a new function is stored in *place*.

5.8.3 Profiling functions and places

Timing information can be obtained for sections of a program. If a program is taking longer than expected to run, or it is suspected to have a bug, examination of the times taken for the execution of different sections might reveal those sections of code which are not time-efficient. These sections can be examined for possible improvements. Profiling information can be obtained both for function calls and for places.

Often it is useful to accumulate profiling information on only the first call of a recursive function, such as the factorial function.

```
(profile (factorial (= call-depth 1)))
```

before

```
(factorial 4)
```

causes timing information for the evaluation of 4! to be accumulated, while:

```
(profile factorial)
```

would cumulatively accumulate the times for evaluation of 4!, 3!, 2! and 1!.

profile [Macro]

Arguments: *{function-description}**

Package: allegro

■ profiles the given functions. *function-description* behaves in the same way as for **trace**. If the functions are traced or breakpointed, or *test-form* is too complicated, inaccurate timings will be produced.

profilef [Macro]

Arguments: *place [test-form]*

Package: allegro

■ profiles the given place. If *place* is traced or breakpointed, or *test-form* is too complicated, inaccurate timings will be produced.

unprofile [Macro]

Arguments: *{function}**

Package: allegro

- removes profiling from the given *function*. If no *functions* are given, all functions and places are unprofiled.

unprofilef [Macro]

Arguments: *place*

Package: allegro

- removes profiling from the given *place*.

profile-reset [Macro]

Arguments: *{function}**

Package: allegro

- resets profile data on the given *function*. If none are provided, all profile information is reset.

profilef-reset [Macro]

Arguments: *place*

Package: allegro

- resets profile data on the function stored at the given *place*.

profile-results [Macro]

Arguments: *{function}**

Package: allegro

- prints to *trace-output* the accumulated profiling data on the *function*, if provided. If not, data on all the profiled functions and places is printed.

profilef-results [Macro]

Arguments: *place*

Package: allegro

- prints to *trace-output* the accumulated data on the given *place*.

Chapter 6 The debugger

The name Debugger is somewhat misleading, as it is a tool which helps you to find bugs, not one which fixes them. The Debugger allows you to examine and modify the current state of the Lisp session, normally after an error has occurred.

When Lisp is started up it is allocated an area of memory in the computer. It divides this area into two main parts, called the heap and the stack. The heap is commonly referred to as the Lisp workspace and will not be discussed further here. The stack is used during the execution of functions to hold control information and transient data. Many books on programming techniques describe the general principles of stacks, their uses and the operations which may be performed on them, so knowledge of this will be assumed throughout the rest of this chapter, which will focus on the Lisp stack and its relation to Lisp code.

6.1 Stack Frames

In most high-level languages, such as Lisp, programs written in a modular way use many separate functions (or procedures) which call each other while the program is executing. At the machine level, the function call mechanism is usually implemented using a stack based approach, which reflects the flow of execution through the program. For instance, if function **A** calls function **B** then function **B** will finish execution before function **A**. This is conveniently implemented by putting the control and data for **A** on a stack, that for **B** on top of it when **B** is called and unstacking it when **B** terminates. The control information for **B** consists of miscellaneous values required to perform the exit from **B** (such as the return address into **A**), while the data is comprised of the values of variables occurring in the Lisp code and anonymous results waiting to be used as part of a computation. Together they are called the stack frame (or activation record) for the particular call to **B**. The Allegro CL Debugger allows you access to these stack frames and their contents.

6.2 Looking at the Stack

To illustrate the Debugger in action, define the following functions:

```
(defun foo (n j)
   (let ((flag (is-ok n)))
     (if flag (foo (1- n)
         (* n j)) (bar j))))
(defun is-ok (x) (> x 1))
(defun bar (j) j)
```

Convince yourself that `(foo n 1)` computes the factorial of n. Now type:

```
(defun bar (j) (error "hi there"))
```

to redefine **`bar`** and call:

```
(foo 3 1)
```

Lisp displays a Restarts dialog box to inform you that an error has occurred (because of the explicit call to **`error`**) and to display the available restarts:

What are restarts?

Allegro CL for Windows 3.0 implements the Common Lisp condition system (see the entry Errors and the Condition System under Lisp Contents in the Online Manual). Conditions (which includes warnings and errors and other things) are signaled and handled by handlers provided by the program or the system. The handler provided by the system for errors dis-

plays a restarts dialog. The user can choose among the various restarts and invoke the one chosen. There are only two restarts in this example, as the illustration of the dialog shows. The user can abort to the Top Level (canceling whatever was being done when the error happened) or enter the debugger. These restarts are almost always available and for that reason buttons equivalent to those restarts are also invoked with the **Abort** and **Enter Debugger** buttons.

What condition is signaled?

Errors as classified according to the condition that was signaled when the error happened. The simplest (and least informative) condition is `simple-error`. A call to **`error`** with a string as its first argument (rather than a condition object) as is done by our function **`bar`** will be a `simple-error`. Many errors in Allegro CL are classified with conditions more specific than `simple-error` but many remain unclassified. If the condition is other than `simple-error`, it is shown in the Restarts dialog. (For example, `(open "foo")` when there is no file called *foo* signals an error of condition `stream-error`.) See the description of the condition system and the function **`error`** in the Online Manual for more information. We do not discuss conditions further in this chapter.

Back to debugging this error

Click the **Enter Debugger** button or invoke the Enter Debugger restart. A backtrace window similar to the following is displayed:

The window title shows the number of active Debuggers (the Debugger can be called recursively). The list on the left contains the stack frames currently active in the Lisp session. The list has one entry for each stack frame, with the most recent frames at the top. The word **Frame** near the top right would identify the number of the selected frame if any frame is selected. Since no frame is selected in the illustration, no number appears.

The buttons on the right identify choices that will be discussed below. Buttons naming unavailable choices are masked and are thus hard to read. The five buttons are, from top to bottom: **More**, **Open**, **Return**, **Restarts**, and **Abort**. (**Return** has no effect in this release of Allegro CL.) The first letter of each is underlined, indicating that pressing the Alt key and that letter is the same as clicking on the button.

The error message associated with the signaled error is shown near the bottom. The condition signaled is also identified unless it is `simple-error` (as it is in this case).

There are several different kinds of frame which are labelled in descriptive ways. When Lisp knows the name of the called function belonging to a particular stack frame it uses that name to label the frame. It has done this for the functions **`debugger`** and **`error`**. foo and bar appear further down and so are not visible in the window.

Seeing more of the stack

There may be many stack frames on the stack, and putting all of these in the scrolling dialog box can be time-consuming, so Lisp initially only fetches some of the frames. Click **More** and (after a pause while Lisp reads more frames) the window is redisplayed. The unhidden frames are mostly system frames. (More has already been clicked in the illustration above).

Entries like `#<function 0 #xB99914>` (near the top of the illustration above) indicates a call to a code object which was not on the **`symbol-function`** of a symbol. This corresponds to a frame created by **`funcall`** or **`apply`**. The `#xB99914` is the address. You will see something different from what is displayed in our picture.

Scroll down the dialog box and then click on the final occurrence of **`foo`** just below the BAR.

This causes the **Open** button to become active and sets the frame number (to 8). Note that we have clicked on **More** already so that button is inactive. (There is no way to show fewer frames.)

> **Frame numbers**. Frame numbers are useful for keeping track of where you are. They start from 0 at the top of the stack.

Copy in the Edit menu is defined to behave in a useful way inside the Debugger window. If you select a frame labelled with a function symbol or a function object, **Copy** will copy the symbol to the Clipboard. **Copy** does nothing if any other kind of frame is selected.

Looking at a stack frame

Double-click on **`foo`** or click **Open** to open up the frame and see what is inside it. A small window is displayed (which you may wish to enlarge) showing the contents of the frame. We have enlarged the window slightly for the illustration.

The name of each argument to the function is given, along with its value. If Lisp does not know the name of an argument, it will label it with a number instead. Other values within the frame will also be displayed, labelled with numbers. In this case, there is one other value, `t`, which corresponds to `flag` in the definition of **`foo`**.

If a function binds special variables, their previous values will be saved on the stack. Double-click the **`debugger`** frame to see an example of this:

This method of binding and unbinding special variables is known as shallow binding. A symbol which is a special variable is bound to a value *V* by saving its old symbol-value on the stack and then setting its symbol-value to *V*.

Copying and changing local variable values. **Copy** and **Paste** work on local variables in stack frames. **Copy** copies the value of the variable; **Paste** changes the variable's value.

Double-clicking on a local variable allows you to inspect the variable's value. You can then use the Inspector to examine and change any data structure to which a local variable is bound. Note the Return Selected Value choice in the right-button menu. It will return the object in the Toploop window (thus making it the value of `*` and allowing you to get a handle on it).

Describing argument values. You can also use **Describe** in the Help menu on the selection to print information about it in the Help window. The selection is first moved to the Clipboard using **Copy**, then the top Clipboard item is popped and passed to **Describe**.

6.3 Exiting From The Debugger

There are several ways of exiting from the Debugger.

Aborting from the Debugger

It is always possible to click **Abort** in the active Debugger window, or close the window by choosing **Close** from the File menu. Both have the same effect. The Debugger calls **`unwind-stack`** with reason `error` to unwind the stack. This behavior is the same as clicking **Abort** in the error dialog box.

Unwinding the stack. This means removing function calls from the stack in a way which maintains the logical consistency of the system. A function exiting normally to its caller is one example of the stack being unwound.

Selecting a restart from the Debugger

The **Restarts** button is always active and, if clicked on, brings up the Restarts window again (it is slightly modified since the Enter Debugger restart displays the current backtrace window, not a new one). The same restarts will be available as when the Restarts window originally appeared.

Returning values

Returning values from a stack frame is not supported in this release of Allegro CL for Windows. The **Return** button on the Debugger window is, therefore, never enabled.

Other notes

Kernel errors. Kernel functions signal errors by calling the function **`sys-error`**. If **`sys-error`** ever exits back to the kernel function, the stack will be unwound as though **Abort** had been clicked in the error dialog box.

Technical note. In your programs, you can exit from the Debugger by executing a **`throw`** or **`unwind-stack`** to a destination below the Debugger on the stack. This causes the stack frames of the Debugger itself to be unwound. As the stack unwinds, the Debugger will restore its entry context using an **`unwind-protect`** form and then resume the unwinding process. The effect of this is to achieve a clean exit, which is usually what was wanted.

6.4 Other Ways Of Entering The Debugger

The Debugger is started by calling the function **`debugger`** with no arguments. You can do this explicitly in your programs. Lisp has several built-in functions and macros which allow your programs to signal errors in a controlled way. Errors can also be signaled by system functions and as a result of you interrupting Lisp.

Interrupting a program

If you think your program is failing, you can interrupt it by pressing the Break (sometimes labeled Break and Pause). You may need to hold the key down for a short time. Lisp will signal a non-continuable error from which you can enter the Debugger and examine the stack. This facility is very useful for when your program gets stuck in a loop.

Recursive Debugger calls

The Debugger may be invoked recursively, for instance, from an error occurring while the Debugger is already running. If this happens, Lisp creates a new Debugger window and hides the previous one. This forces you to exit from the most recent Debugger first. The previous Debugger will not attempt to restore its entry context before the new Debugger is invoked. On exit from the new Debugger, the old Debugger window appears again.

6.5 Debugger Preferences

There is no choice for Debugger on the Preferences menu.

The Printer preferences include `*trace-print-level*` and `*trace-print-length*` which are used to control output from the **`backtrace`** function, and also output from the Trace utility.

> **Backtracing**. The **`backtrace`** function uses the same rules for suppressing frames. When called it prints a description of the current state of the Lisp stack to the stream `*debug-io*`, which is initially a synonym stream of `*terminal-io*`.

6.6 Debugger Reference

Stack frame labels

Within the backtrace window you may see frames labelled in a variety of ways. The meanings of these labels are given in the following table:

`FOO` *(or similar)*	An executing call to the Lisp function of the same name.
`#<function...>`	An executing call to a Lisp function made by **`funcall`**, **`apply`** etc.
`#<closure...>`	An executing call to a closure made by **`funcall`**, **`apply`** etc.
`open FOO` *(or similar)*	A call to the Lisp function of the same name which has not yet had all its arguments evaluated.
`open #<function...>`	A call to a function which has not yet had all its arguments evaluated.

`open #<closure...>`	A call to a closure which has not yet had all its arguments evaluated.
`catch` *`tag`*	A catch frame with tag *`tag`*.

Controlling the Debugger

Here are the effects of the buttons in a Debugger window.

More	fetches more frames from the stack (no undo).
Open	opens a window showing the contents of the selected frame.
Return	returns values from the selected frame (not supported in this release of Allegro CL).
Restarts	redisplays the restarts window (allowing invocation of available restarts).
Abort	unwinds the stack with reason **`error`**

Closing a Debugger window is equivalent to clicking **Abort**.

6.7 Debugger internals

This section was chapter 7 of the *Inside Programming Tools* manual in release 2.0.

Debugging facilities are available from the Toploop and via a Lisp function interface. The Lisp stack is divided into frames, corresponding to invocations of Lisp constructs such as lambda expressions and function calls. The convention used throughout is that the top of the stack is the most recent frame. It is possible to exclude sequences of functions or packages from the action of the Debugger, and to vary the volume of output produced.

6.7.1 Debugger external functions

debugger [Function]

Arguments: `&optional` *`continuable-p`* `t`

Package: `allegro`

■ enters the debugger. No values are returned, and the function may not always return. If *`continuable-p`* is `nil`, the function will not return, but will exit using an `(unwind-stack nil 'error nil)`.

backtrace [Function]

Arguments: `&key (:frame-count 10000) (:style :locals)`

Package: `allegro`

■ prints the top *`n`* frames of the stack to the stream `*debug-io*`. If *`n`* is omitted or is `nil`, the whole stack is printed. *`style`* can take one of three values:

- `:fns` if only functions are to be printed
- `:args` if both functions and arguments are to be printed
- `:locals` if both functions and arguments are to be printed, along with locals.

6.7.2 Default window sizes

The first four variables control the size of the debugger windows that pop up on the right edge of a backtrace window, to show the arguments to a function call.

`*variable-browser-min-width*` [Variable]

Package: `debugger`

- The initial value is 100.

`*variable-browser-max-width*` [Variable]

Package: `debugger`

- The initial value is 200.

`*variable-browser-min-height*` [Variable]

Package: `debugger`

- The initial value is 100.

`*variable-browser-offset*` [Variable]

Package: `debugger`

- The initial value is 11. This variable controls the vertical distance between the tops of successive windows.

The next four variables control the size and position of backtrace windows. Multiple browsers will be placed one on top of another. If you move a browser window, these variables will be reset to correspond to the new location so the next browser will come up there. The left and top coordinates are 50 and `nil` initially, meaning place in lower left, above the status bar if it is present, 50 pixels from left edge.

`*stack-browser-window-left*` [Variable]

Package: `debugger`

- Initial value 50. The value of this variable is changed if a browser is moved by the user (with, e.g. the mouse) to reflect the new position.

`*stack-browser-window-top*` [Variable]

Package: `debugger`

- Initial value `nil`, which (new in release 3.0) means place the window near lower left of `*lisp-main-window*`. `nil` allows the system to take the presence or absence of the status bar into account when placing the browser so that no portion of the browser is obscured by the status bar. The value of this variable is changed if a browser is moved by the user (with, e.g. the mouse) to reflect the new position.

`*stack-browser-window-width*` [Variable]

Package: `debugger`

- Initial value 352.

`*stack-browser-window-height*` [Variable]

Package: `debugger`

- Initial value 286.

Chapter 7 The stepper and the watch facility

Using the Stepper, you can single step the evaluation of any Lisp form. This can be useful to find the source of an error when the other debugging tools such as Trace and Breakpoint have not produced any helpful information. The Stepper works by augmenting the source code used by the compiler - it does not use an interpreter.

The Watch facility allows you to make Lisp print the value stored in any **`setf`**able place every time the Toploop prints its prompt. The values being watched are displayed in a special window.

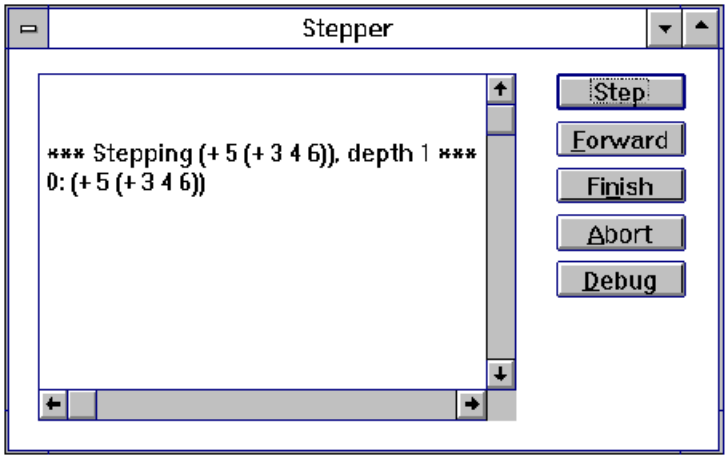
7.1 The Stepper

Stepping a form

To step a form, you use the macro **`step`**. For example, click on the Toploop window and type:

```
(step (+ 5 (+ 3 4 6)))
```

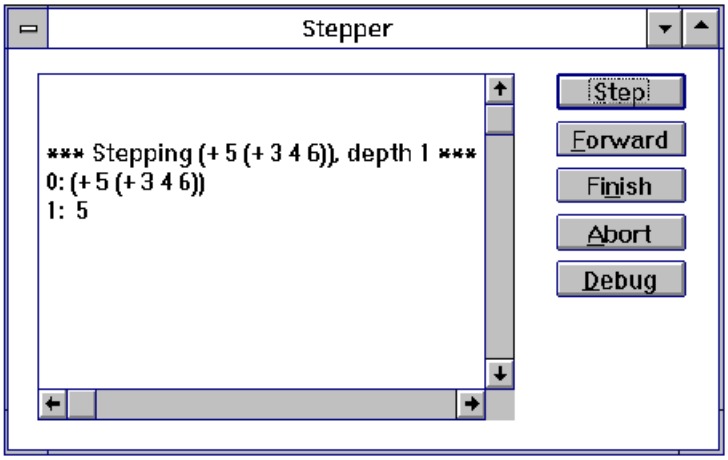
Press Enter. Lisp displays the Step Control window:



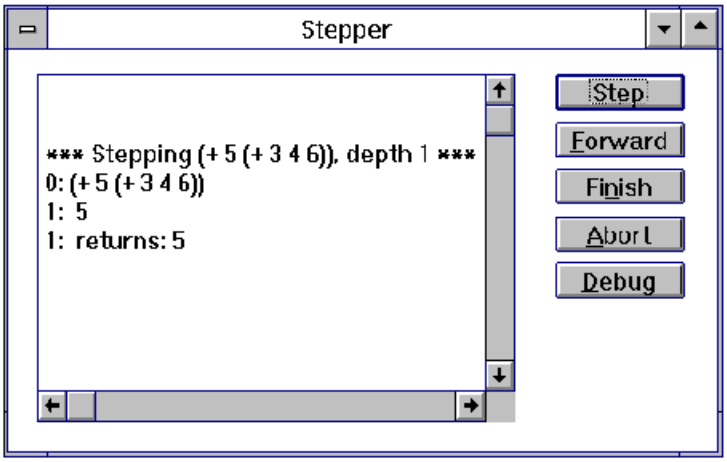
The Step Control window, which we also call the Stepper window (from its title), contains buttons which allow you to control the single stepping of the form displayed in the Stepper window. The second line in the Stepper window shows the subexpression about to be evaluated and its level of nesting in the whole expression.

Simple stepping

Click **Step** in the Step Control window to single step the evaluation of `(+ 5 (+ 3 4 6))`. Lisp displays the lexical level (nesting) and the subexpression it is about to evaluate:



Click **Step** again and Lisp displays the result of evaluating that subexpression.



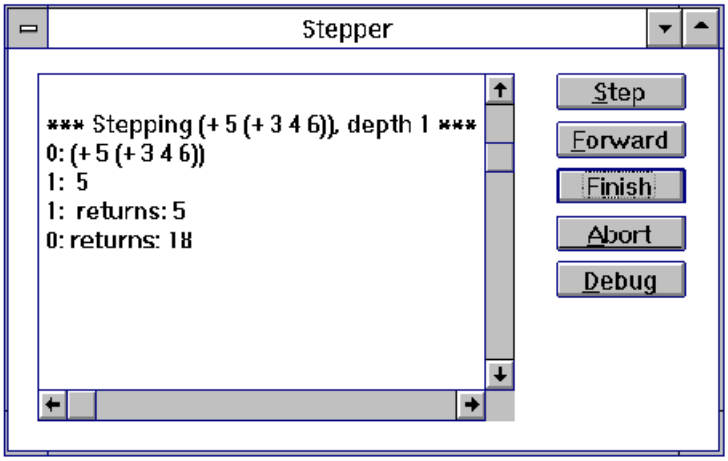
You can click **Step** repeatedly to carry on stepping through the evaluation of `(+ 5 (+ 3 4 6))`.

Skipping to the end

You can quickly skip to the end of the evaluation of a stepped form by clicking **Finish** in the Step Control window. For example, click **Abort** in the Stepper window, then enter, in the TopLoop window:

```
(step (+ 5 (+ 3 4 6)))
```

and click **Step** twice, then click **Finish**. Lisp displays the result returned by the whole form:



Click **Step** again and the stepping operation is completed, returning control to the Toploop again.

Finish does not end stepping. Clicking **Finish** does not complete the stepping operation. This allows you to enter the Debugger just before the stepped expression returns. This is useful when you are using Step other than as a top-level form.

Aborting stepping. You can abort stepping of a form at any time by clicking **Abort** in the Step Control window. Control is immediately returned to the Toploop.

The Stepper window. The Stepper window is not closed when the stepping operation is completed. You can print the Stepper window or save it to disk. The

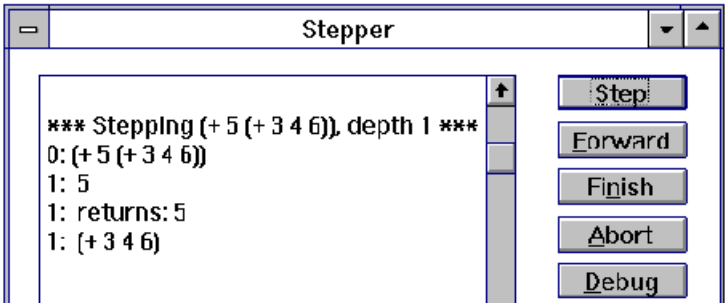
Stepper is a Text Edit window so you can type to it. Note however, you cannot modify text to be evaluated while stepping and type in text to be evaluated.

The Toploop window while stepping

While you are stepping, the form containing step that you typed to the Toploop window is still being evaluated. Therefore, you cannot type anything further to the Toploop window until stepping is completed. You can always quit stepping by clicking on the **Abort** button.

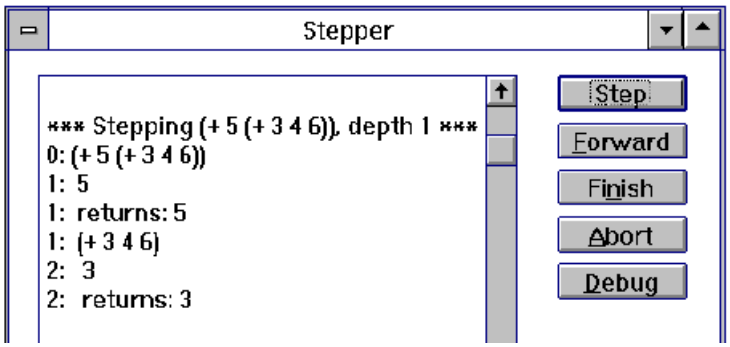
Skipping subexpressions

Using **Forward**, you can evaluate a subexpression without having to step through all the expressions it contains. For example, if necessary, click **Abort** in the Stepper window and enter `(step (+ 5 (+ 3 4 6)))` and click **Step** three times. Lisp displays the form it is about to evaluate:

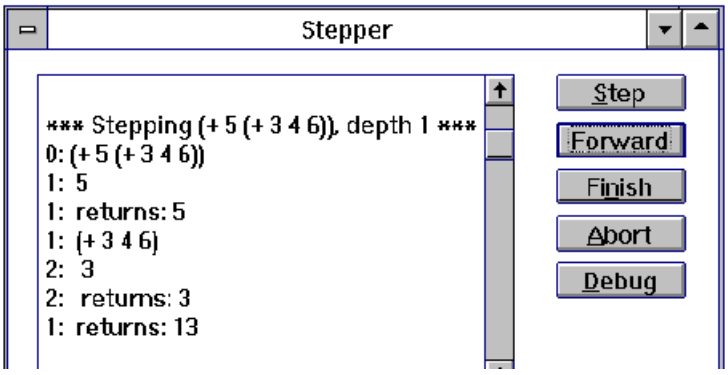


Now click **Forward**. Lisp displays the result returned by `(+ 3 4 6)` without stepping through all the subforms in it. (To see the difference, try clicking **Step** instead of **Forward** in the example above).

You can also skip forward to the end of any layer of subexpression nesting. As an illustration of this, click Abort in the Stepper window, then enter, in the TopLoop window, `(step (+ 5 (+ 3 4 6)))` and click **Step** five times. Lisp displays the result returned by evaluating 3, which is nested two levels deep in the stepped form:



Now click on the line containing `1: (+ 3 4 6)` in the Stepper window and click **Forward** in the Step Control window. Lisp displays the result returned by evaluating the subexpression `(+ 3 4 6)` without any further stepping of the subexpressions within it:



Click **Step** again and stepping continues from the end of the evaluation of `(+ 3 4 6)`.

Selective stepping. Using a combination of **Forward** and **Step**, you can achieve a very high degree of control when stepping a form.

Entering the Debugger

You can enter the Debugger any time during the stepping of a form by clicking **Debug** in the Step Control window. This can be useful to find out from where a stepped function has been called. You can exit from the Debugger and continue stepping by clicking Continue in the Debugger window. Clicking **Abort** in the Debugger window aborts the stepping operation.

Saving the Stepper Output

The Stepper window can be saved to disk by choosing **Save** from the File menu.

7.2 Using step other than at the Top Level

`step` can be used anywhere within a Lisp program, not just as a top-level form. For example, you can define a function as:

```
(defun foo (x) (step (+ 5 (+ 3 4 x))))
```

and the Stepper is only invoked when foo is called, not when it is compiled.

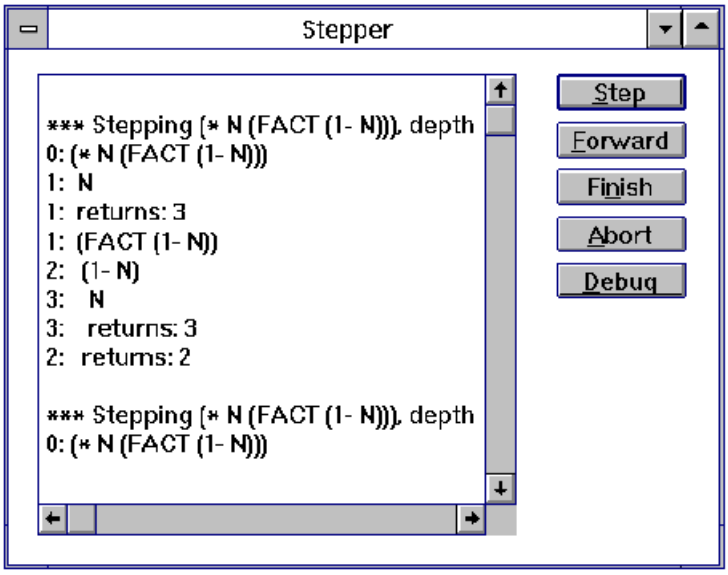
> **Fine control over stepping**. Using **`step`** in this way, you can step only the parts of your code that you need to, and restrict the need for stepping to a minimum.
>
> **Shortcut**. In the Structure Editor, you can easily insert a call to **`step`** around the selected structure by choosing **Break At...** from the Tools menu and then clicking **Step** on the submenu that is displayed.

Stepping recursive functions

You can step within recursive functions, and each recursive call will be stepped. For example, define **`fact`** as:

```
(defun fact (n) (if (<= n 1) 1 (step (* n (fact (1- n))))))
```

Now type `(fact 3)`. Lisp displays the Stepper window and the Stepper window. Click **Step** to step through the expression `(* x (fact (1- x)))`. After you have clicked **Step** eight times, Lisp displays a message to show that **`step`** has been called again in the recursive call to **`fact`**:



Finding out where `step` has been called from

You can use the **Debug** button in the Stepper window to invoke the Debugger to see where `step` has been called from.

Skipping over recursive calls to `step`

Clicking **Forward** skips over recursive calls to `step` in subexpressions.

7.3 Watching Places

Lisp allows you to monitor the value stored in any **`setf`**able place.

Starting watching a place

Click in the Toploop window and type:

```
(watch *print-length*)
```

Lisp displays a new window titled Watch showing the value stored in the place, which in this case is the symbol-value of the symbol `*print-length*`.

Now type:

```
(setq *print-length* 3)
```

and Lisp updates the value of the place displayed in the Watch window. Lisp will carry on updating the Watch window every time the Toploop prompt is printed.

Shortcut. You can set up watching several places at once by giving more than one argument to **`watch`**.

Finding out which places are being watched. You can get a list of all the places currently being watched by typing `(watch)`.

The Watch window. You can print the Watch window or save it to disk, just like any other Text Editor window.

Stopping watching a place

You can use the macro **`unwatch`** to stop a place being watched. Click in the Toploop window and type:

```
(unwatch *print-length*)
```

to switch off watching of that place.

Shortcut. You can unwatch several places at once by giving more than one argument to **`unwatch`**. You can unwatch all watched places in one go by typing `(unwatch)`.

Technical note. You can update the Watch window from your programs using the function **`allegro:watch-print`**, which takes no arguments.

Chapter 8 CLOS tools

Allegro CL for Windows provides several tools for examining CLOS objects. There are four tools available, corresponding to the four items on the **CLOS Tools** submenu of the Tools menu.

There are toolbar buttons for the four choices, but they are not on the default toolbar. If you want them on the toolbar, bring up the toolbar palette and add them.

Browse Class

Selecting **Browse Class** from the submenu displayed by the CLOS Tools item on the Tools menu when a symbol naming a class is selected brings up the following non-modal dialog.

The left side displays a graph of superclasses of the class of interest (in this case text-edit-window, which was selected when we brought up this dialog). The right side is a tab-control widget with tabs for supers (shows the direct superclass), subs, methods, slots (one for direct and one for all), and the class precedence list.

In the graph, an open folder means there are subclasses (which may or may not be displayed) while a page-with-folded-corner means there are no subclasses. Clicking on the plus (+) at a graph node displays more of the graph while clicking on a minus (-) displays less. The right button menu over a selected item in the graph allows further investigation of the selected class. Note the **Class Commands** choice, which displays a submenu of class-related choices. Single clicking on an item in the graph displays more graph.

Look at the status bar

The status bar is often updated as you move the mouse about the Class Outline dialog. You may wish to increase the number of lines in the status bar to see all the information (choose **Status Bar Number of Lines** from the right button menu over the status bar)

Graph subclasses and Graph superclasses

You can choose these from the **CLOS Tools** submenu. They display graphs of the sub- or super-classes of the class being browsed. We show these graphs in the next illustration. The superclasses is above.

The items in the graph are mouse sensitive. If you right click over a graph entry, the following menu is displayed:

It provides the usual choices. The menu was displayed over the `stream` entry, and `stream` is the first item. Choosing it displays an inspector window on the `stream` class.

If you right click over the background of the graph, a menu appears with items Options, Recompute Graph, and Print Graph (to the printer). Recompute Graph recomputes the graph so it reflects changes you have made. Options displays this dialog:

The **`Initial depth`** box allows you to restrict the graph to a specified number of levels. `nil` means no restriction.

Browse Generic function

This too can be chosen from the menu or from the box on the browser dialogs. Choosing it displays the following dialog:

Edit Class, Edit Method, and Edit Generic Function

These boxes causes the file where the object is defined to be displayed, opening the file to the definition of interest. (Since we have been browsing system-supplied objects, the system would tell us that it cannot find the definitions if we chose these options.

Remove Method

This choice removes the selected method.

Chapter 9 Comtabs

Command tables (or *comtabs*) provide a way of calling a particular Lisp function automatically when a virtual-key-down event or sequence of events occurs. They are often used in the implementation of editors to associate Lisp functions with particular key combinations. A command table is automatically associated with every Lisp edit window you open, and you can also use comtabs to handle events in any Lisp stream built on `comtab-mixin` if you wish. This often means that you can avoid writing your own event loop function for the stream.

Comtabs are built up in three steps:

- **`make-comtab`** (Section 9.1) is used to build an empty comtab with a defined inheritance path.
- The comtab functions are defined.
- The event or menu selection bindings are established with **`set-event-function`** (Section 9.2).

9.1 Defining comtabs

It is not usually necessary to define all the commands in a comtab, because the inheritance mechanism means that you can use definitions from existing tables. The default parent for new comtabs is `comtab:*default-comtab*` which handles only very basic events. `text-edit:*raw-text-edit-comtab*` provides some text entry facilities in addition.

`comtab-mixin` `[Class]`

Package: `comtab`

■ supplies the comtab slot and event function so that streams built with this mixin can look up functions from virtual-key-down events.

make-comtab [Function]

Arguments: &key (:inherit-from comtab:*default-comtab*)

Package: comtab

■ creates and returns a comtab which inherits from :inherit-from, or *default-comtab* if no inheritance is specified.

comtab-p [Function]

Arguments: *object*

Package: comtab

■ returns t if *object* is a comtab, otherwise returns nil.

set-comtab [Function]

Arguments: *stream comtab*

Package: comtab

■ associates the comtab *comtab* with *stream* and returns *comtab*. In the following example, a new comtab, my-comtab is created and associated with the text edit window, my-window. my-comtab inherits from *raw-text-edit-comtab*.

```
(setq my-window
    (open-stream 'te::lisp-editor-window
       *lisp-main-window* :io
       :title "Comtab example window"))
(setq my-comtab
    (comtab:make-comtab
        :inherit-from te:*raw-text-edit-comtab*))
(comtab:set-comtab (selected-window my-window) my-comtab)
```

comtab [Function]

Arguments: *stream*

Package: comtab

■ returns the comtab associated with *stream* or nil if there is none.

inherit-from [Function]

Arguments: *comtab*

Package: comtab

■ returns the parent of *comtab* or nil if there is none. This function is **setf**able.

illegal-operation [Macro]

Arguments: *stream*

Package: comtab

■ should be called by a comtab function if an invalid operation is attempted. **illegal-operation** beeps and returns nil.

Function, etc. to display comtab bindings

The following function will generate a table displaying commands available in a specified set of comtabs.

comtab-report [Function]

Arguments: (&optional (*filename* "comtabs.txt")
(*comtab-names* *comtab-names*)
(*max-column-width* 20)

Package: comtab

■ Generates a table showing all of the editor commands available in some set of comtabs, with names of the keystrokes that map to these commands.

■ *filename* should be a string naming a file. The report will be written to that file.

comtab-names should be a list of symbols whose values are comtabs to be included in the report. Note that each comtab is given a column, so reporting on too many will generate a report that is too wide.

max-column-width is the maximum number of text columns any field of the report may occupy.

comtab-names [Variable]

Package: comtab

■ The default set of comtabs that will be included in the report, if the *comtab-names* argument is not passed to **comtab-report**. The initial value is

```
(te::*host-comtab* te::*emacs-comtab* te::*brief-comtab*)
```

These are the comtabs for the three editor modes.

9.2 Event functions

set-event-function [Function]

Arguments: *comtab events function*

Package: comtab

■ adds a command to *comtab* for handling *events*, which may be either a single event or a list of events. Event processing is described in detail in the *Common Graphics* section on the Online Manual. If more than one event is given, *function* will be called if they occur sequentially, separated only by null events. *function* must take one argument, the stream in which the comtab *event* occurred. The special variables *event-number*, *button-state*, *cursor-position* and *time* described below in Section 9.3 are bound before *function* is called. **set-event-function** returns *function*.

As an example, the following code segment defines a key binding for #\Control-\[in my-comtab, assuming that it inherits from text-edit:*raw-text-edit-comtab*. Typing Control-[in a window using my-comtab calls **print-definition**, which moves the cursor to the start of the enclosing definition, reads a Lisp form and prints it to *standard-output*.

```
;;; First define the function to read
;;; a definition from "window" and print it.


(defun print-definition (window)
    (let ((*read-no-hang* t))
        (text-edit:beginning-of-definition
            window)
```

```
        (print (read window nil nil))))

;;; Now add the event function to the comtab.

(comtab:set-event-function my-comtab
    '(control-key pc:vk-open-square-bracket)
    #'print-definition)
```

event [Generic Function]

Arguments: *(stream comtab-mixin) event buttons data time*

Package: comtab

■ This is the event function used by text edit windows to convert *events* in a Lisp *stream* into calls to the appropriate comtab functions. It is automatically inherited by text edit windows. Comtab event processing may be implemented in any Allegro CL for Windows stream by using this function to handle events. See the *Common Graphics* section on the Online Manual for general details of event handling.

9.3 Event variables

The only argument passed to a comtab event function is the stream in which the event occurred. The following special variables are provided to allow comtab functions to make their action dependent on the event which caused the function to be called, the mouse button state, the cursor position and the time. See the *Common Graphics* section on the Online Manual for details of the format of these variables. Their values are only valid during execution of a comtab event function. They may be destructively modified by the system when the next event is processed, so you must make a copy of any you want to keep.

event-number [Variable]

Package: comtab

■ the number of the event which caused the comtab function to be called.

button-state [Variable]

Package: comtab

- the state of the mouse button and modifier keys when the comtab function was called.

`*cursor-position*` `[Variable]`

Package: `comtab`

- the cursor position when the comtab function was called.

`*time*` `[Variable]`

Package: `comtab`

- the time when the comtab function was called.

Appendix A Editor keybindings

The following table shows keybindings for the Emacs, Brief and Host text editor modes. Note tat additional keystrokes are available in Host mode. These are the ones shown in the keyboard shortcuts in menubar items.

Command	**Emacs**	**Brief**	**Host**
Apropos	Ctrl-A	---	---
Backward Character	Ctrl-B	---	←
Backward Delete Word	---	Ctrl-Backspace	---
Backward Kill Line	Alt-K	---	Alt-Ctrl-Backsp
Backward Kill Sexp	Ctrl-Alt-Backsp	---	Ctrl-Backsp
Backward Kill Word	Ctrl-Backsp	---	---
Backward List	Ctrl-Alt-P	---	---
Backward Sexp	Ctrl-Alt-←	---	Alt-←
Backward Up List	Ctrl-Alt-↓	---	---
Backward Word	Alt-←	---	Ctrl-←
Beginning Of File	Shft-Alt-Comma	Home	---
Beginning Of Line	Ctrl-A	---	---
Beginning of previous definition	---	---	Alt-↑
Beginning Of Next Definition	Ctrl-Alt-]	---	Alt-↓
Capitalize Word	Alt-C	---	---
Complete symbol	Control-.	Control-.	Control-.
Delete Horizontal Space	Ctrl-Slash	---	---
Delete Indentation	Ctrl-6	---	---
Delete Line	---	Alt-K	Alt-Delete
Delete Next Character	Ctrl-D	---	Delete
Delete next S expression	---	---	Alt-Delete
Delete Previous Character	Ctrl-H	---	Backsp
Delete To Kill Buffer	Ctrl-W	---	---
Delete Whole Line	---	Alt-D	---
Delete Word	---	Alt-Backspace	---
Describe	Ctrl-X Ctrl-I	---	---
Documentation	Ctrl-X Ctrl-D	---	---
Down List	Ctrl-Alt-D	---	---

Command	Emacs	Brief	Host
Downcase Word	Alt-L	---	---
Edit File	Ctrl-X Ctrl-F	---	---
End Of Definition	Ctrl-Alt-E	---	---
End Of File	Shft-Alt-Period	End	---
End Of Line	Ctrl-E	---	---
Eval	Ctrl-X Ctrl-E	Alt-F10	---
Eval Clipboard	---	---	---
Eval Definition	Ctrl-Alt-C	---	---
Exchange To Mark	Ctrl-X Ctrl-X	---	---
Find	Ctrl-S Alt-S	F5	---
Find Same	---	Shft-F5	→
Find Toploop Prompt	Ctrl-Shift-N	Ctrl-Shift-N	Ctrl-Shift-N
Forward Character	Ctrl-F	---	---
Forward List	Ctrl-Alt-N	---	---
Forward Sexp	Ctrl-Alt-→	---	Alt-→
Forward Up List	Ctrl-Alt-)	---	---
Forward Word	Alt-→ Alt-F	---	Ctrl-→
Illegal Operation	Ctrl-G Ctrl-X Ctrl-G Alt-G Ctrl-Alt-G	---	---
Indent For Comment	Ctrl-Semicolon	---	---
Insert Empty List	Alt-9	---	---
Insert Empty String	Shft-Alt-2	---	---
Kill Comment	Ctrl-Alt-;	---	---
Kill Line	Ctrl-K	---	---
Kill Sexp	Ctrl-Alt-K	---	---
Kill Word	Alt-D	---	---
Lambda List	Ctrl-X Ctrl-L	---	---
Load File	Ctrl-X Ctrl-R	Alt-R	---
Macroexpand	Ctrl-X Ctrl-M	---	---
Mark	---	Alt-M	---
New	Ctrl-X Ctrl-B	---	---
Newline	Enter Ctrl-M	---	Ctrl-Enter
Newline And Indent	Ctrl-J	Enter	Enter
Next Line	Ctrl-N	---	---
Next Page	Ctrl-V	---	---
Open	---	Alt-E	---
Open Line	Ctrl-O	Ctrl-Return	---
Paste	---	Insert	---
Previous Line	Ctrl-P	---	---
Previous Page	Alt-V	---	---

Command	Emacs	Brief	Host
Quick lambda list	Ctrl-Z	Ctrl-Z	Ctrl-Z
Quick lambda list and insert space	space	space	space
quit	---	Alt-X	---
Reindent Region	Ctrl-X Ctrl-Q	Tab	---
Reindent Sexp	Ctrl-Alt-Q	---	---
Reindent Single Line	Tab	---	---
Replace	Ctrl-R	F6	---
Replace Same	---	Shft-F6	---
Replace Without Dialog	Alt-R	---	---
Return To Reader	Enter	Enter	---
Save	Ctrl-X Ctrl-S	Alt-W	---
Save As	Ctrl-X Ctrl-W	Alt-O	---
Scroll One Line Down	---	Ctrl-D	---
Scroll One Line Up	---	Ctrl-U	---
Scroll To Selection	Ctrl-L	---	---
Select All	Ctrl-X A	---	---
Select To Mark	Ctrl-Space Ctrl-Space	---	---
Set Mark	Ctrl-2	---	---
Transpose Characters	Ctrl-T	---	Ctrl-‘
Transpose S expression	---	---	Alt-‘
Undo	Ctrl-X Ctrl-U Alt-U	*	---
Upcase Word	Alt-U	---	---
Yank From Kill Buffer	Ctrl-Y	---	---
Zoom Window	Ctrl-X Z	---	---

[This page intentionally left blank.]

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

[This page intentionally left blank.]

Allegro CL for Windows

General Index

version 3.0

October, 1995

Copyright and other notices:

This is revision 1 of this manual. This manual has Franz Inc. document number D-U-00-PC0-12-51019-3-1.

Copyright © 1994, 1995 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademarks of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

PREFACE

This document is a general index for all the bound manuals included with the Allegro CL for Windows product. Each entry in this index identifies the manual, the volume (I or II), the chapter in the manual, and the page in the chapter. (Chapter pages are numbered N-M where N is the chapter number and M is the page number within the chapter. Note that the *Getting Started* manual does not have chapters so its pages are number 1, 2, 3, etc.)

We have simply combined the indexes for each manual.

The table below page identifies the manuals from the codes used in this index. The codes are also explained at the foot of every page of index.

Code	Manual	In Vol.
CLI	*Common Lisp Introduction*	1
FFI	*Foreign Function Interface*	1
GS	*Getting Started*	1
IB	*Interface Builder*	1
PT	*Programming Tools*	2

Each manual has its own index as well.

Note that the *Professional* supplement, which includes the *Runtime Generator* manual, (only provided with the Professional version of Allegro CL for Windows) is not indexed in this document.

[This page intentionally left blank.]

Index

Symbols

Numerics

A

CLI - *Common Lisp Intro* (I)
GS - *Getting Started* (I)
PT - *Programming Tools Intro* (II)
FFI - *Foreign Function Interface* (I)
IB - *Interface Builder* (I)

B

C

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

D

E

F

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

G

H

I

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

J

K

L

CLI - *Common Lisp Intro* (I)
GS - *Getting Started* (I)
PT - *Programming Tools Intro* (II)
FFI - *Foreign Function Interface* (I)
IB - *Interface Builder* (I)

M

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

N

O

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

P

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

Q

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

R

S

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

T

CLI - *Common Lisp Intro* (I)
GS - *Getting Started* (I)
PT - *Programming Tools Intro* (II)
FFI - *Foreign Function Interface* (I)
IB - *Interface Builder* (I)

CLI - *Common Lisp Intro* (I)
GS - *Getting Started* (I)
PT - *Programming Tools Intro* (II)
FFI - *Foreign Function Interface* (I)
IB - *Interface Builder* (I)

U

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

V

CLI - *Common Lisp Intro* (I)
FFI - *Foreign Function Interface* (I)
GS - *Getting Started* (I)
IB - *Interface Builder* (I)
PT - *Programming Tools Intro* (II)

W

Y

Z

[This page intentionally left blank.]

Allegro CL for Windows

Professional Supplement

version 3.0

October, 1995

Copyright and other notices:

This is revision 0 (initial version) of this manual. This manual has Franz Inc. document number D-U-00-PC0-09-51020-3-0.

Copyright © 1992, 1994, 1995 by Franz Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, by photocopying or recording, or otherwise, without the prior and explicit written permission of Franz Incorporated.

Restricted rights legend: Use, duplication, and disclosure by the United States Government are subject to Restricted Rights for Commercial Software developed at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

Allegro CL is a registered trademarks of Franz Inc.

Allegro CL for Windows is a trademark of Franz Inc.

Windows, MS Windows, MS-DOS, and DOS are trademarks of Microsoft.

Franz Inc.
1995 University Avenue
Berkeley, CA 94704
U.S.A.

Contents

Index

Chapter 1 Introduction to the Professional version

The Professional version of Allegro CL 3.0 for Windows is designed for people interested in serious software development and in delivering applications to others. While the Professional version includes additional functionality (compared to the Standard version), the most important addition is the Runtime Generator, which allows you to produce standalone applications, and a license to distribute those application free of charge.

This document briefly describes the additional functionality and services provided with the Professional version. The longest chapter is chapter 4, which describes the Runtime Generator. The chapters in this manual are:

1. **Introduction to the Professional version**. The chapter you are now reading.
2. **Support and source code**. All Professional customers are entitled to customer support as detailed in this chapter. They can also obtain sources for certain parts of Allegro CL for WIndows, as detailed in this chapter.
3. **Sockets and the Grid widget**. The Professional version contains code for interprocess communication (via sockets). It also contains code for the Grid widget. Both are described in this chapter.
4. **Runtime Generator**. The Runtime Generator allows you to create standalone applications (based on your Lisp programs) which you can distribute royalty free. All aspects are described in this chapter.

There is an index at the end of the manual.

Standard version customers can purchased the various additions available in the Professional version individually. This document may be sent to Standard version customers who have purchased one or more additions. Those customers should note that this manual describes all possible add-ons even though they may have only purchased some of them.

Chapter 2 Support and source code

2.1 Support

All Professional version customers are entitled to product support from Franz Inc. (for a period of time specified in the license agreement -- check with your sales representative for information on the time limits and renewal options of product support).

In this section, we provide some details of support and make some suggestions on how you can help Franz Inc. in supporting you.

Send in the Source Code and Support addendum

You will find a document entitled *SOURCE CODE AND SUPPORT ADDENDUM TO FRANZ INC. ALLEGRO CL 3.0 FOR WINDOWS PROFESSIONAL SOFTWARE LICENSE AGREEMENT* included with the distribution. You must fill out, sign, and return this document in order to be eligible for support.

Contacting by telephone or FAX

When you have a problem, you can call for telephone support (+ 510-548-3600) during regular business hours (8:00 AM to 5:00 PM United States Pacific time -- 8 or 9 hours west of Greenwich Mean Time depending on the time of year -- every working day -- Monday through Friday except holidays). Please note that you may have to hold or leave a number where you can be called back if support personnel are otherwise engaged.

Or you can send a FAX anytime (+ 510-548-8253). The information in a FAX should be the same as sent by e-mail, which we describe below.

Best way to contact: e-mail to pc-support@franz.com

The most efficient and most effective way to obtain customer support for Allegro CL for WIndows is to send electronic mail. This method is preferred for the following reasons:

- Mail (like faxes) can be sent anytime of the day or night.
- A response can be sent at anytime as well.
- All customer support electronic mail is seen by all development personnel at Franz Inc. This means that the expert who will deal with the problem (who may be different from the telephone support person) will see the message at once.
- Your words and your description of the problem are seen by the expert who handles the problem. (That person is often different from the telephone support person.)
- Examples and code samples will be machine readable (a big advantage over faxes). Note that it is virtually impossible to transcribe code samples over the telephone.

The email address for Allegro CL 3.0 for Windows support is **pc-support@franz.com**. Use this address for *any* question, comment, or request for technical information. (Requests for sales and product information are best sent to **info@franz.com**.)

A pcspr

When you send in a question, bug report, or problem description, it will be assigned a pcspr number (pcspr stands for PC Software Problem Report). The entire record of communication on the issue will be stored and can be referenced with the pcspr #. Please be sure to include that number in the subject line of any e-mail and in the body of any other message. Support personnel will then know what you are referring to and will be able to look at the record to see what has already happened on the issue.

But please do not bring up new questions, bugs, or problems referencing an existing pcspr #. Instead, send a message without a pcspr reference. It is best if each pcspr deals with one issue (or a few related issues).

How to report a bug or problem

Before reporting a bug, please study the documentation to be sure that what you experienced is indeed a bug. If the documentation is not clear, this is a bug in the documentation: Allegro CL may not have done what you expected, but it may have done what it was supposed to do.

A report that such and such happened is generally of limited value in determining the cause of a problem. It is very important for us to know what happened before the error occurred: what you typed in, what Allegro CL printed out. A verbatim log, may be needed. If you are able to localize the bug and reliably duplicate it with a minimal amount of code, it will greatly expedite repairs.

It is much easier to find a bug that is generated when a single isolated function is applied than a bug that is generated somewhere when an enormous application is loaded. Although we are intimately familiar with Allegro CL, you are familiar with your application and the context in which the bug was observed. Context is also important in determining whether the bug is really in Allegro CL or in something that it depends on, such as the operating system.

To this end, we request that your reports to us of bugs or of suspected bugs include the following information. If any of the information is missing, it is likely to delay or complicate our response.

- **Lisp implementation details**. Tell us the version of Allegro CL for Windows that you are using, including at least the release number and date of release (printed in the banner when Allegro CL comes up), and the operating system and its release number (Windows 95, NT, 3.1 or Windows for Workgroups). The make and manufacturer of your PC is sometimes helpful, as is information on non-standard features which you have added, if any. We also need to know what patches are loaded (the value of `acl:*patches*`) and what modules are loaded (the value of `*modules*`). The contents of the *allegro.ini* file are often helpful. That file is located in the directory where Allegro CL was installed.
- **Information about you**. Tell us who you are, where you are and how you can be reached (an electronic mail address if you have one, a postal address, and your telephone number), your Allegro CL serial number (displayed by choosing **About Allegro CL** from the Help menu, also on the CD jewel box).
- **A description of the bug**. Describe clearly and concisely the behavior that you observe and what you expected, if it is different.

- **Exhibits**. Provide us with the *smallest, self-contained* Lisp source fragment that will duplicate the problem, and a log (e.g. produced with **dribble**) of a *complete* session with Allegro CL that illustrates the bug.

A convenient way of generating at least part of a bug report is to use the **dribble** function mentioned above. Typing

```
(dribble filename)
```

causes implementation and version information to be written to the file specified by *filename*, and then records the Lisp session in the same file. Typing

```
(dribble)
```

will close the file after the bug has been exhibited. **dribble** is defined in the Online Manual. The following dialogue provides a rudimentary template for the kernel of a bug report.

```
> (dribble "bug.dribble")
> (pprint acl:*patches*)  ;; what patches are loaded
> (pprint *modules*) ;; what modules are loaded
> ;; Now duplicate your bug . . .
> (dribble)
```

Send the resulting file, along with the additional material asked for above and we will investigate the report and inform you of its resolution in a timely manner.

We will meet you more than halfway to get your project moving again when a bug stalls you. We only ask that you take a few steps in our direction.

2.2 Source code

Professional version customer are entitled to the sources for certain parts of the Allegro CL for Windows system. These sources are not included with the distribution because they are separately licensed. When you have sent in the license addendum described just below, the relevant sources will be sent on a diskette.

Send in the Source Code and Support addendum

You will find a document entitled *SOURCE CODE AND SUPPORT ADDENDUM TO FRANZ INC. ALLEGRO CL 3.0 FOR WINDOWS PROFESSIONAL SOFTWARE LICENSE AGREEMENT* included with the distribution. You must fill out, sign, and return this document in order to be eligible for the source code, which will be sent to you on a diskette after the addendum has been received by Franz Inc.

What sources are distributed

Sources for the following are sent upon receipt of the license addendum:

- Common Graphics
- The Interface Builder
- The Text Editor
- The Grapher
- The MCI interface
- The Grid Control Widget

Installation instructions will be sent with the source code diskette telling you how to integrate the sources with the Find Definition facilities of the Text Editor.

Do not load the source files

The sources are for facilities either already present in Allegro CL 3.0 for Windows, or available as a loadable module. (The Grid Control Widget is a loadable module.) You are strongly discouraged from loading these sources into the Lisp system, whether modified or not functionality (unless instructed to do so by Franz Inc.) Modifying these features may break things in ways that you do not expect, and (because our version will be different from yours), we will have difficulty provided customer support to deal with problems that arise.

[This page intentionally left blank.]

Chapter 3 Sockets and Grid Widget

3.1 Socket interface

The Professional version of Allegro CL 3.0 for Windows includes a socket interface for facilitating interprocess communication. It is described online (in the Online Manual, under the heading **Sockets**) and in *ex\socket\socket.txt*.

You must load socket.fsl

The socket interface code is not included by default in the Lisp image. In order to have the functionality available, evaluate the following form:

```
(require :socket "fsl\\socket.fsl")
```

You may wish to put this form in your *startup.lsp* file so it will be loaded automatically when Lisp starts up.

Files in the distribution

- *fsl\socket.fsl* -- the fsl file containing the facility
- *ex\socket\socket.txt* -- an online description of the interface
- *ex\socket\apserver.lsp* -- an examples file
- *sockaux.dll* -- (in the Allegro CL distribution directory) a DLL loaded into Lisp when needed.

winsock.dll is needed

If you are running Windows 95 or Windows NT, you should already have a *winsock.dll* provided by MicroSoft. It will be loaded automatically when it is needed.

If you are running Windows 3.1, there is no *winsock.dll* file provided by default be MicroSoft, so you will need to obtain one, usually from a third party. One such is Trumpet Winsock (v1.0 at the time of printing). This is shareware available (again, when this manual was printed) from

```
inorganic5.chem.ufl.edu:/gopher/pub/winsock/
```

If you are running Windows for Workgroups, again there is no *winsock.dll* provided by default but MicroSoft provides a TCP/IP stack which includes a *winsock.dll* file. At the time of printing, information on this file (and others) could be obtained from:

```
ftp://ftp.microsoft/com/Softlib/index.txt
```

sockaux.dll must be distributed with applications

If you have the Runtime Generator and are thus licensed to distribute applications generated by that product (see chapter 4 of this document), you may use socket communication in your application. If you do so, be sure to distribute *sockaux.dll* along with the other files you distribute with your application.

3.2 The Grid widget

The Grid is a rather complex widget that displays as a rectangular array of cells, and can be used to implement applications such as spreadsheets. A grid can have multiple sections of rows and columns, where each section may be independently scrolled.

Grid documentation

The Grid widget is documented in the Online Manual under the heading **Grid-widget**.

You must load the various grid*.fsl files

The Grid widget code is not included by default in the Lisp image. There are three files that you can load: *fsl\grid.fsl* (which has the basic functionality); *fsl\griddev.fsl*, puts the Grid widget in the Interface Builder; and *fsl\gridtest.fsl*, which is a Grid example.

In order to have the functionality available, evaluate the following forms:

```
(require :grid "fsl\\grid.fsl")
(require :griddev "fsl\\griddev.fsl")
(require :gridtest "fsl\\gridtest.fsl")
```

You may wish to put these forms in your *startup.lsp* file so it will be loaded automatically when Lisp starts up.

The gridtest example is initiated with the form `(grid-example)` after loading *grid.fsl* and *gridtest.fsl*.

Note that the Interface Builder only knows about the grid widget after *grid.fsl* and *griddev.fsl* are loaded.

[This page intentionally left blank.]

Chapter 4 Runtime Generator

4.1 Introduction

The Runtime Generator is used to convert programs requiring the support of the Allegro CL for Windows standard system into programs that can be executed independently. Such programs are called *standalone applications* and they rely upon the support of the Allegro CL for Windows runtime system to execute.

The conversion process analyzes which parts of the full Lisp system are required to support the application and removes unused parts. This makes the resulting application smaller and therefore less demanding on memory and disk space. Against this, it requires that your code be entirely self-supporting since inadvertent references to absent objects are almost certain to cause your program to fail.

Although the procedure for creating a standalone application is largely automatic, it can be quite lengthy. In practice, it is not worth generating the application until you are satisfied that the code has been thoroughly tested and debugged while running in the full Allegro CL programming environment: **Producing a standalone application should be the very last step of program development**.

Suitable applications

The Runtime Generator is specially intended for applications that do not use parts of the standard system such as

- the compiler
- the **`eval`** function
- programming tools

This permits a considerable reduction in image size. Some of your programs may make use of the Allegro CL development environment: the programming tools or Lisp compiler, for example. Unless you avoid these dependencies, it is only feasible to use such programs in conjunction with the standard Lisp system.

The eventual space savings depend on how much of the standard Lisp system needs to be retained in order to support completely your standalone application. For example, if you do not use **`format`**, the Runtime Generator can be instructed to delete that part of the Lisp system and the resultant image will be smaller. For optimum savings, you have to assist the Runtime Generator by providing information on what may be safely deleted, as explained in section 4.3.

Licensing

Allegro CL for Windows Professional version licensees are granted the right to distribute, royalty-fee, applications which were generated using the Allegro CL for Windows Runtime Generator. The file *lisp.exe* may be distributed royalty-free as well, and the files *aclgut16.dll*, *aclgut32.dll*, and *sockaux.dll*. (The *aclgut*.dll* files are for the universal thunk. The *sockaux.dll* file is needed for interprocess communication via sockets. All are in the distribution directory. They may not be required, but you may want to distribute them anyway with your application just to be safe.) Franz Inc. needs to be contacted for any other distribution arrangements.

Installation

All Runtime Generator files are included in the Professional Allegro CL for Windows distribution. The runtime image *runtime.img* is installed as part of the installation process so just follow the standard installation instructions found in the *Getting Started* document. *runtime.img* has a program item installed along with the normal Allegro CL program item. It is labeled 'Allegro CL Runtime'.

Using the runtime generator

To make fullest use of the facilities and options provided by the Runtime Generator requires a working knowledge of Lisp and those parts of the language your application uses. Nevertheless, provided you are aware of the basic principles, it is relatively straightforward to use the Runtime Generator with its default settings. The result may not be the best that can be achieved, but it is possible to produce a working standalone application, gain experience of the operation of the system and get some idea of the likely space savings.

The first part of this section recapitulates the purpose, scope and definitions associated with the Runtime Generator which have already been outlined in the section 4.1. As a demonstration, the last subsection takes you through the production of a simple standalone application called Lunar Lander working from source code provided in the distribution. Other examples are also referred to in that section.

4.1.1 The Runtime Generator

The Runtime Generator is used in conjunction with the Allegro CL for Windows system to create standalone applications which use the smaller Allegro CL runtime system. The Generator has three components: the runtime-system Lisp application, a Lisp file called *readme.lsp* in the *ex\runtime* directory and the runtime-system image that was created during the installation procedure.

4.1.2 Standalone applications

A standalone application consists of at least two parts: the runtime-system Lisp application and a Lisp image made by loading your entire source code into the runtime-system image.

The permanently resident part of the standalone application may be smaller than 200kB. The only thing the standalone application can do is run your program; it cannot be used as a general-purpose programming tool and it is impossible to add new functions to it.

4.1.3 The Runtime System

The runtime-system Lisp image (*runtime.img*) is a cut-down version of the Allegro CL standard image. It is intended as a stepping stone to a Lisp image incorporating your source code. As a consequence, its programming environment and development facilities are poor. Most of the programming tools (including the Debugger) are absent, or present in only a rudimentary form. The text editor provides only basic operations of a general nature and most of the menus provided in the standard system are not available. However, it does include the full compiler (although the compiler is removed as part of the runtime generation process) and many of the Common Lisp functions and symbols.

4.1.4 Making a standalone application

The process of generating a standalone application is quite straightforward. In outline, you startup Allegro CL Runtime, load in your Lisp source code, then call a single function to create the standalone image. Finally, you create a program item for the application.

This function can take a quite complex set of arguments so a template is included in the *ex\runtime\readme.lsp* Lisp file supplied as part of the distribution. You can edit a copy of this code and then load it into the system after your source code to initiate the application creation. The listing also contains comments to assist you in choosing the arguments appropriate for your application. A dialog box for creating a call to **`create-standalone-application`** is also provided, as described in section 4.3.5.

In this section, we make a sample application to get the feel of what should happen when things are going right.

4.1.5 Lunar Lander example

The *ex\runtime* directory contains a file called *lunaland.lsp*. This is a very simple demonstration program along the lines of the text-based Moon landing programs popular on teletypes before the advent of cheap graphics terminals. The game can be run by loading the

code and then typing `(program)`. Type in a thrust value between 0 and 30 to reduce your rate of fall towards the Moon and hit Enter to see how the lander responds. Landing with a velocity of less than 10 (in unspecified units) is considered a success.

To make a standalone version of Lunar Lander, start Lisp by double-clicking on Allegro CL Runtime. Then load in *ex\runtime\lunaland.lsp* using the **Load** option on the File menu. Now load *ex\runtime\mklunar.lsp* and tell Lisp where to save the standalone application. Wait while Lisp does some internal tidying up and a few garbage collections. It then saves the application to the place you specified. You can create a program item for Lunar Landing by following the instructions at the end of section 4.3.1.

To run the Lunar Lander application, double-click on the program item. Lisp starts up and the Lunar Lander window appears. To quit the application you can close the window, or play the game and then decline to play another, or interrupt the program using the **Break** key (sometimes called **Break and Pause**).

Other examples

The *ex\runtime* directory also contains a bouncing smiley-face example (the files *smiley.lsp* and *mksmiley.lsp*) and a prototype call to **`allegro::create-standalone-application`** with comments and advice (*readme.lsp*). The smiley-face example is discussed in section 4.3.6 below.

4.2 Making an application

Once you have developed, tested and debugged your intended application using the Allegro CL for Windows standard system, you can move on to the production of a standalone version. It must be emphasized that it is only worth proceeding when you are sure that your code is thoroughly debugged; much grief can ensue otherwise!

To make a standalone application, start Allegro CL Runtime (it was installed as part of the Allegro CL for Windows installation and its program item should be next the Allegro CL program item).

Once you have started Allegro CL Runtime, there are four steps to producing your application:

1. Load any needed modules.
2. Add a function to launch your application.
3. Make your code less reliant upon the Allegro CL programming environment.
4. Proceed by steps to remove parts of the runtime system, checking at each stage that the resultant application executes satisfactorily.

There are plenty of things which can go wrong and it is wise to proceed with caution. By adopting a methodical approach, it is easier to backtrack in the event of a problem.

Step 4 takes place in two stages. First, parts of the Lisp system are removed automatically. This action is always performed and you cannot override it. Next, additional parts are optionally removed according to your instructions when you initiate the application creation. Guidance on precisely what is done and how to proceed is given in this section. We use the Smiley-Face application mentioned in Section 4.2.5 as an example to illustrate the creation of an application. Common problems are described in section 4.4.

4.2.1 Making your program standalone

Once you are satisfied that your code works correctly in the Allegro CL standard system, you can make the changes needed for it to run on its own. This usually involves writing some additional support code.

Creating an image requires starting up Allegro CL Runtime and calling the function **`allegro::create-standalone-application`**. The file *readme.lsp* contains a template for a call to that function. In this section and section 4.3.2 and 4.3.3, we discuss what the call will look like, telling you how to modify a copy of the file *readme.lsp* so it contains the correct call. The dialog displayed by choosing **Create Standalone Form** from the **Images** submenu of the File menu can be used either to create a file containing a call to **`allegro::create-standalone-application`** or makes the call itself. See section 4.3.5 **Using the Create Standalone Form dialog** for more information.

Conditionalizing for the runtime generator

At times, you may find that you need to make the behavior of your source code conditional on whether it is being evaluated (and therefore compiled) under the standard Lisp system or the runtime system. For example, the runtime version may have to specially build a menu that would have been automatically present under the standard system. Rather than maintain two versions of your code, you can use the conditional read macros `#+` and `#-` to respectively invoke or disable code specific to runtime-system evaluation. `#+` and `#-` are followed immediately by a feature name and then a form which is evaluated conditionally on the presence or absence of this feature on the list held by `*features*`. (See the descriptions of `#-` and `#-` and of `*features*` in the Allegro CL Online Documentation.) The Runtime Generator includes the feature `:runtime-system`. For example, in the following code:


```
(for directory bound *sources-directory* and filename in
    '("file1"
      #+runtime-system "file2"
      #-runtime-system "file3")
    do
    (load (merge-pathnames filename directory)))
```

file1 is always loaded, *file2* is only loaded when the code is being evaluated in the runtime system, and *file3* is loaded unless the code is being evaluated in the runtime system.

Loading modules that are needed

Developers of standalone applications are pulled in two directions: They want functionality available in case it is needed and they want the application to be small. Striking a balance is never easy. Later in this section, we describe how things are thrown out of the image when a standalone application is created. Some thing, however, are left out of the image at the beginning and need to be loaded in if they will be needed.

The file *readme.txt* in the Allegro CL directory (by default, *c:\allegro*, the directory where Allegro CL was installed) contains commented-out forms that load particular modules into a Lisp image (loading is done with the function **`require`**). You should copy this file to a *lsp* file (say *rg-mods.lsp*), uncomment the require forms for the features needed in your application, and load the modified *rg-mods.lsp* into the Runtime image. This will cause the uncommented modules to be loaded and will make the associated functionality available.

What follows is a partial list of the modules that are available (look at the *readme.txt* file for the complete list). Note that the ones associated with the Grid widget and the Socket interface are supplied only with the Professional version of Allegro CL for Windows. (This document may be provided to Standard version customers who purchased the Runtime Generator add-on but who may not have purchased the Grid or Socket add-ons. All Professional version customer have all listed modules.)

Note too the Grid Widget for the Interface Builder cannot be loaded into a Runtime and so should not be uncommented.

```
;; REQUIRED BY ALL IN-LISP WIDGETS (OUTLINE, GRID,
;; AND LISP-GROUP-BOX)
; #P"fsl\\lispwij.fsl"
;; THE OUTLINE CONTROL
; #P"fsl\\outline.fsl"
;; THE GRID-WIDGET
; #P"fsl\\grid.fsl"
;; THE GRID WIDGET IN THE INTERFACE BUILDER (CANNOT LOAD
;; INTO ALLEGRO RUNTIME)
; #P"fsl\\griddev.fsl"
;; THE GRID EXAMPLE
; #P"fsl\\gridtest.fsl"
;; A GROUP-BOX WHICH IS 3D EVEN IN WINDOWS 3.1
; #P"fsl\\groupbox.fsl"
```

```
;; REQUIRED BY ALL WINDOWS 95 COMMON CONTROLS
; #P"fsl\\comcon.fsl"
;; HEADER-CONTROL COMMON CONTROL
; #P"fsl\\header.fsl"
;; PROGRESS-INDICATOR COMMON CONTROL
; #p"fsl\\progress.fsl"
;; THE MULTI-SECTION COMMON-STATUS-BAR COMMON CONTROL
; #P"fsl\\statbar.fsl"
;; TAB-CONTROL COMMON CONTROL
; #P"fsl\\tab.fsl"
;; TRACKBAR COMMON CONTROL
; #P"fsl\\trackbar.fsl"
;; UP-DOWN-CONTROL COMMON CONTROL
; #P"fsl\\updown.fsl"
;; DYNAMIC DATA EXCHANGE (DDE)
; #P"fsl\\dde.fsl"
;; [... remainder of excerpts from readme.txt deleted. Look at
;; the readme.txt file for more information and complete list of
;; modules.]
```

If you try to use functionality that is not loaded, you get the usual undefined function, unbound variable, etc. errors just as if you tried to use functionality in any unloaded file. because most of the functionality is in the Lisp image (but not in the Runtime image), you may want to start by uncommenting everything and then cutting back.

The entry point to your application

When a standalone application is launched, it performs some initialization and then calls a function whose effect is to run your program. This means your program must be written to have a single entry point - a Lisp function taking no arguments - which can be called to execute your program. The Runtime Generator must be informed of the name of this function when you make the standalone application.

The process of producing a standalone application is initiated by calling **`allegro::create-standalone-application`** in the runtime system. The function takes an extensive list of arguments and you may find it helpful to edit the template provided in the *ex\runtime\readme.lsp* file supplied in the distribution (edit a copy of the file). The edited template can then be loaded into the runtime system directly after loading your application source code.

To illustrate, suppose you have loaded your source code into the runtime system which has defined your startup function as **my-program** in the common-lisp-user package. (In other words, if you wanted to, evaluating (my-program) would launch your program.) The first executable form in a copy of the file *ex\runtime\readme.lsp* should be changed to specify this:

```
(allegro::create-standalone-application
    'common-lisp-user::my-program
    ...)
```

Loading the modified form will commence production of the standalone application, using **my-program** as the entry point.

allegro::create-standalone-application takes one compulsory argument, the name of your program's entry point, and several keyword arguments which specify which parts of the system may be removed and control aspects of the resulting standalone application. These all default to the most conservative value, so initially you can omit them all. This will result in a fairly large application containing everything that it is possible to retain.

Once your code works in this environment you can change some of the arguments to remove parts of the system that you know you do not actually need but that Lisp would hold on to by default just in case. For example, it is frequently possible to delete some of the packages, but Lisp must keep them by default in case you try to access or intern symbols in them.

Having successfully defined your single entry point, you may need to go on and make additional changes to take over operations formerly performed by parts of the system which are always removed, itemized in the next section.

> **Note**: **allegro::create-standalone-application** takes a while to execute. As part of this execution, it clears the screen. (On Windows 3.1, only the frames of various windows are visible.) Do not fear! Once the standalone application has been created, the Runtime Generator itself exits (and, under Windows 3.1 control will be returned to Windows). Note too that creation of the standalone application is memory-dependent, and takes longer on machines with less memory. We do not recommend creating a standalone application on a machine with only 4 Megabytes of memory since the process can over an hour.

Creating a program item

`allegro::create-standalone-application` creates an image file (for example *c:\myapp\foo.img*). Now you may want to create a shortcut for the application (under Windows 95) or a program item for the application in the Program Manager (under the old Windows shell).

To create a shortcut under Windows 95, assuming your Allegro CL for Windows kernel is in *c:\allegro\lisp.exe*, do the following:

1. Right-click in desktop and select **New** and then **Shortcut**.
2. For command, type

   ```
   c:\allegro\lisp.exe c:\myapp\foo.img
   ```

3. For name, choose a name and enter it.

Under Windows 3.1 or using the old Windows shell, do the following, again assuming your Allegro CL for Windows kernel is in *c:\allegro\lisp.exe*:

1. In the Windows Program Manager, click on the File menu and select **New**.
2. In the New Program Object dialog box, select **Program Item**.
3. In the Program Item Properties dialog box, type:

   ```
   c:\allegro\lisp.exe c:\myapp\foo.img
   ```

   into the Command Line space.

   The item will be displayed with the Allegro CL for Windows icon unless the Program Item icon is changed by selecting **Change Icon** from the Program Item Properties dialog box. Change the icon if you desire.

This will create the Program Item and completes creation of your standalone application. The application can now be copied and distributed (with the Allegro CL for Windows kernel file *lisp.exe*).

4.2.2 Objects always removed

While creating a standalone application, some actions are carried out automatically by the Runtime Generator. They are listed below along with comments about the implications and what to do in case of difficulty. The information given here is necessarily of a technical nature.

- **The compiler is removed**. This is a fundamental limitation on the kind of program that can be made into a standalone application. It also destroys **`eval`** and prevents the `#.` and `#,` read macros from working.
- **All macros are removed**. You do not need them without the compiler.
- **The `proclaim` function is disabled**. It is only useful at compile time and the compiler is not present. Removing it saves some space.
- **The About Application" dialog box is redefined**. This is the dialog box you normally see when you select the last item of the Help menu of an application, e.g., with the standard Allegro CL for Windows image, the box is entitled "About Allegro CL for Windows". If you define a function **`common-lisp-user::about-application`** in your code then it will be called with no arguments when the box is selected in the standalone application. Otherwise selecting the **About Application** box will do nothing.
- **Definition and documentation information is removed**. Most of such information present in the Allegro CL for Windows standard system is already absent from the runtime system. Documentation strings and lambda-list information is also removed from the user's code; it is unnecessary in an executing standalone application.
- **Some session-init-functions are removed**. Those initially present in the runtime system are removed. Any added later are retained in the correct order, so that the first one added by your code (at loadtime or runtime) will be the first in the list. This is unlikely to affect you unless your running program uses the list. The session-exit-functions and session-variables lists are similarly modified.
- **Some comtabs are modified**. All menu commands and command names are removed from the comtabs `*default-comtab*`, `*raw-text-edit-comtab*` and `*text-edit-comtab*`. If you want menu commands or command names in those comtabs, you must add them yourself at runtime.

- **The `toploop` package is destroyed**. Type information, structures, function definitions, property lists and values of symbols in the `toploop` package are removed. The package is then deleted. Therefore you should avoid putting your own symbols into the `toploop` package. At runtime the Toploop window is not opened and `*terminal-io*` is not bound to an open stream. Unless you take steps to the contrary, any output to `*terminal-io*` will cause an unrecoverable error. Since warnings, printed error messages and the like are sent to this stream, you may want to open it explicitly yourself. If you want to see printed messages and so on, bind it to an open window. If you want all text sent to it to just disappear, bind it to a null stream, as created by `(make-broadcast-stream)`. You may also need to set up `*error-output*` as a synonym stream of `*terminal-io*` to handle error messages.
- **Some variables are re-initialized**. `*print-case*` and `*read-case*` are set to `:upcase`.
- **The Online Manual is removed**. The Allegro CL Online Manual describes Common Lisp.

A note on the unavailability of eval

Because **`eval`** is not available in a Runtime image, you cannot construct a form and evaluate it. However, you can use **`funcall`** or **`apply`** to achieve the same result. Suppose you want to construct and evaluate the form

```
(foo arg1 arg2)
```

Either of the following two forms, both of which will work in a Runtime image (assuming that **`foo`** is defined as a function, of course), are equivalent:

```
(apply #'foo (list arg1 arg2))
(funcall #'foo arg1 arg2)
```

4.2.3 Controlling the Runtime Generator

You can obtain further savings in the size of standalone applications by making use of the keyword arguments passed to **`allegro::create-standalone-application`**. If they are not required, removal of the error handlers, **`format`**, the printer and the reader produces a significant reduction in size.

It is necessary to leave the extent of removal to the programmer's discretion since many system functions check their arguments and call an error handler if they detect a problem. Note however that the standard error handlers use **format**, and **format** uses the printer. If you remove the error handlers you may need to add **trap-exits** and **errorset** forms to allow your program to die gracefully in the event of an error. Otherwise an error being raised will result in a call to **unwind-stack** and an untimely exit from your application.

The keywords of **allegro::create-standalone-application** are listed in the table below along with a description of their effect:

Table 1: Keyword arguments to create-standalone-application

Keyword Argument	Description
`:classes-to-remove`	Specifies which CLOS classes to remove. The classes are provided as a list. The default is `nil`.
`:image`	Specifies the pathname for the resulting standalone application. If `nil` (the default) Lisp prompts with a file dialog; clicking Cancel in the dialog aborts application creation.
`:image-part-size`	A numeric value that specifies the maximum size of the image file in bytes. If the size of the image exceeds the specified size, the image is split into parts to conform to the specification. If `nil` (the default), no maximum size is specified and the image is not split.
`:interruptible`	`nil` if Break key break handling is to be disabled within the standalone application. The default is `t`, i.e. interrupts enabled. Note that an interrupt (under certain conditions) may cause your program to crash.

Table 1: Keyword arguments to create-standalone-application

Keyword Argument	Description
`:no-dialog-on-errors`	If non-`nil` the error handlers **`error`**, **`sys-error`** and **`break`** are redefined to call **`unwind-stack`** with arguments `nil`, `error`, `nil`. **`warn`** will do the same if `*break-on-warnings*` is non-`nil`. Unless you take preventative measures, any errors will cause the application to exit.
`:remove-clos-lambda-lists`	Specifies whether CLOS lambda lists are to be deleted. If `nil` (the default), the lambda lists are not removed.
`:remove-format`	If non-`nil`, **`format`** is removed. Trying to call **`format`** will cause an error. The default is `nil`.
`:remove-printer`	If non-`nil`, the printer is removed. Trying to call the printer will cause an error. The default is `nil`.
`:remove-reader`	If non-`nil`, the reader is removed. Trying to call the reader will cause an error. The default is `nil`.
`:package-directives`	An association list controlling the processing of packages. Each element of the list should be a dotted pair of the form `(item . directive)`. `item` is coerced to a package and must therefore be an actual package or something acceptable to **`find-package`**. See just below for more information.

More on :package-directives

As said in the last table entry above, the value of the `:package-directives` argument must be an association list whose elements are dotted pairs of the form `(item . directive)`. `item` must identify a package. The list of possible `directive`s and their

effects is given in the table below.

Directive	Effect
`:keep`	leave the package alone
`:internal`	delete unreferenced internal symbols
`:external`	delete unreferenced internal and external symbols

Deleting a package does not delete its symbols, but makes them uninterned. If you need to keep uninterned symbols, e.g. because you have attached properties to them or given them values, you must either reference them from your code (or know that they are referenced from system code which is referenced from your code) or use one of the directives above. However, if you use **`read`** or **`intern`** and want them to find symbols correctly, you must arrange to keep the appropriate packages.

- **If the value of `:package-directives` is not `nil` and a package does not appear in any package directive it is explicitly deleted. The `toploop` package is always subject to an explicit `delete-package`.**
- By default, the value of the `:package-directives` keyword argument is `nil` meaning all packages except the `toploop` package are processed as though they had directive `:keep`.

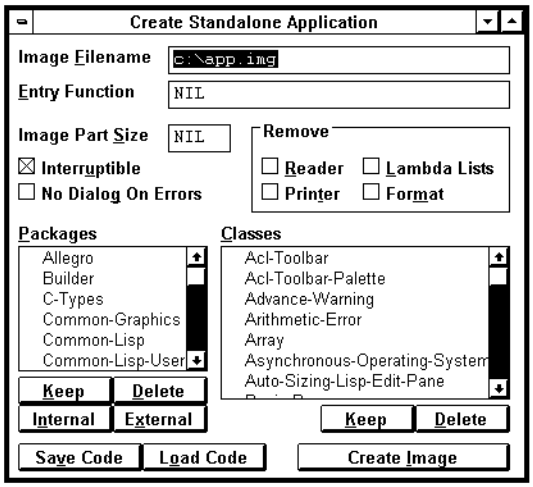
4.2.4 Recommended Development Path

Once you have reached the point where your standalone application executes correctly using the default keyword arguments of **`allegro::create-standalone-application`**, you may wish to consider making further savings. We suggest you add keyword arguments in the following order. After each step you should check that the standalone application still works. If it does not, refer to section 4.4 for guidance on corrective action. If size is an important consideration, you may find it worthwhile rewriting your application to avoid using one or more of **`format`**, the printer and reader, which can then be removed by the use of keywords.

1. Add `:image` and `:interruptible` keywords. These are unlikely to cause problems.
2. Eliminate the reader if you do not need it. This should not cause problems but be aware that the reader is required by some dialog box items e.g. those with the widget `lisp-text`.
3. Eliminate the error handlers, **`format`**, and the printer if you do not need them. This should not cause additional problems but any runtime errors will not be reported in dialog boxes; instead your program will abort. If you leave the standard error handlers but remove **`format`** or the printer, an error causes stack overflow due to a sequence of "Undefined Function" errors while trying to report the original error.
4. Delete some packages and remove symbols from others using the `:package-directives` argument. There is a form in the Runtime Generator file *readme.lsp* which computes the packages defined by you or used by one of your packages: you may find this useful. Unless you specifically need to read or intern a symbol from a package other than one of those, you will probably find that specifying a `:external` directive to all the packages computed by the form results in a working standalone application. You must always keep the `system` and `user` packages but a directive of `:external` for these packages will normally be acceptable.

4.2.5 Using the Create Standalone Form dialog

Choosing **Create Standalone Form** from the **Images** submenu of the File menu displays the following dialog.



Each argument to **`allegro::create-standalone-application`** has an associated widget allowing you to specify it in the dialog. After you have made your selections, click on **Create Image** to create an image file (you must be running Allegro CL Runtime for this to work). Click on **Save File** to save the call to **`create-standalone-application`** in a file. **Load File** will load a file with such a call, allowing you to modify the arguments before **`create-standalone-application`** is actually called. You need not be running Allegro CL Runtime for **Save File** and **Load File** to work.

4.2.6 Example of creating your own application

This section provides a step-by-step description of the creating of the Smiley-Face standalone application, once the application code itself had been written and debugged completely. This is done to illustrate the process described in the preceding sections.

The code for the Smiley-Face application is found in the file *ex\runtime\smiley.lsp*. We will illustrate in this section how the associated application-generation file, *mksmiley.lsp*, was created. Each step of the process described below includes the following seven sub-steps:

(a) Start the Runtime Image of Allegro CL for Windows. You will normally load modules specified in *readme.txt* at this point, but no modules are required for the Smiley Face application. See the information under the heading **Load modules that are needed** in section 4.3.1 above for more information.

(b) Edit a file to create the application (*mksmiley.step#* -- i.e. *mksmiley.1*, *mksmiley.2* etc.)

(c) Load in the application code, *smiley.lsp*

(d) Load in the creation code, *mksmiley.step#* (i.e. *mksmiley.1*, *mksmiley.2* etc.)

(e) After the application has been built, create the Program Item

(f) Ensure that the application works correctly

(g) Check the size of the resulting image (You can view the resulting image size by using the File Manager -- click on the **View** option and select **Partial Details**. In the dialog box for **Partial Details**, select the **Size** option.)

For brevity in the description below, we will not explicitly repeatedly mention sub-steps (a), (c), (d), (e) and (f), as the operations involved are repetitive and identical with each iteration through this loop. We discuss only sub-steps (b) and (g), i.e., the editing of the creation file, *mksmiley.**, and the size of the resulting image. (For details about creating a Program Item for the standalone application, refer to section 4.3.1.)

Step 1:

(b) Create a file, *mksmiley.1*, specifying the entry point for the standalone application, with the form:

```
(allegro::create-standalone-application 'user::smile)
```

When prompted for a name for the resulting image, type "smiley.1".

(g) The resulting size will be only marginally smaller than the Runtime Image itself.

Step 2:

(b) Copy *mksmiley.1* to *mksmiley.2*. In *mksmiley.2*, add the keywords for `:image` and `:interruptible`, using (change *allegro*\ appropriately if you loaded the distribution into another directory):

```
:image "c:\allegro\ex\runtime\smiley.2" :interruptible t
```

(g) The resulting size will be virtually unchanged from Step 1.

Step 3:

(b) Copy *mksmiley.2* to *mksmiley.3*. In *mksmiley.3*, add the keyword for `:remove-reader`, i.e., specify:

```
:remove-reader t
```

(g) Again, the resulting size will be virtually unchanged from Step 1.

Step 4:

(b) Copy *mksmiley.3* to *mksmiley.4*. In *mksmiley.4*, add the keywords for errors and printing, i.e.,:

```
:no-dialog-on-errors t :remove-format t :remove-printer t
```

(g) Again, the resulting size will be virtually unchanged from Step 1.

Step 5:

(b) Copy *mksmiley.4* to *mksmiley.5*. In *mksmiley.5*, add the keyword for package-directives, i.e.,:

```
:package-directives '((:common-lisp-user . :external)
                      (:system . :external)
                      (:printer . :external)
                      (:common-graphics . :external))
```

Note that we do not include the `allegro` package because the symbol for **`create-standalone-application`** is internal to this package (if `allegro` was added to the list above, an error would result during application creation). Note also that specifying the printer package actually increases the size of

the image slightly, because all of the printer package had earlier been removed (using the `:remove-printer` keyword).

The `(let (old-package-list ...))` form in *readme.lsp* can be used at this point to determine which packages are used by the application. In the case of this application, no additional information was made available by evaluating this form.

(g) The resulting size should be significantly smaller than the previous sizes.

Step 6:

(b) Copy *mksmiley.5* to *mksmiley.6*. In *mksmiley.6*, add the keywords for CLOS, i.e.:

```
:remove-clos-lambda-lists t :classes-to-remove '()
```

(g) The resulting size of the image should be smaller than the image in step 5.

Using the steps above, the image size is typically reduced to about 40% of its original size. While other applications may not be reduced as much the example does illustrate the savings available through relatively mechanical operations. Further savings can be achieved through a better understanding of the application and the underlying system, but they will require more time than the steps above.

4.3 Common Lisp Object System

CLOS Restrictions

As explained in Section 4.1, standalone applications cannot use the Lisp compiler. This means that the CLOS functions **`add-method`**, **`remove-method`**, **`ensure-generic-function`**. which recompile the generic function, cannot be used.

Some types of method combination call **`compute-effective-method`** which in turn calls the compiler. They are:

- generic functions with optimizations off (e.g. speed level 0 or 1);
- user-defined long-form method combination;
- user-defined short-form method combination where the operator is a special form or macro.

Note: Actually, a code template scheme is used to avoid compiling every time. In cases where the effective method code is **`equal`** to a previously called effective method (treating all calls to **`call-method`** as **`equal`**), the compiler is not invoked. Thus, to use the above method combination types in a runtime application it is necessary first to fill the cache by calling generic functions with these method combinations with all of the patterns of arguments with which they will later be called.

Local generic functions created by **`generic-flet`**, **`generic-function`**, **`generic-labels`** and **`with-added-methods`** are created in a different way without calling the compiler. However, they will call **`compute-effective-method`** (see above).

4.4 Troubleshooting

Section 4.3 covered some of the problems you might encounter while trying to create a standalone application. This section deals with difficulties that may be encountered whilst trying to run the application. Only the more common problems are described, though this should be sufficient to indicate where the trouble lies.

In general, runtime errors are more tricky to track down as some detective work is involved. As the programmer, you are at a disadvantage since you can no longer call upon such facilities as the Allegro CL Debugger. You may also have directed the Runtime Generator to remove **`format`** and the Lisp printer, so unanticipated Lisp errors cannot be reported without causing more errors and the system quickly ties itself into a knot. In many cases, you cannot recover from the error and you will be forced to exit from the standalone application giving you no opportunity to investigate further.

Most solutions are based on the strategy of going back a step or two in the development process until you produce a version of your intended standalone application that works. You should also double-check that your code functions correctly in the Allegro CL for Windows standard system and that your application is tailored for turning into a standalone application.

The remainder of this section is divided into two subsections suggesting causes of specific errors and more general problems that arise.

4.4.1 General Problems

➨ Too many garbage collections occur

Fault:

- There is very little free memory when Lisp is running.
- The program is generating lots of garbage.

Fixes

- Try to remove more from the standalone application.
- Increase the `Heapsize` variable in the *allegro.ini* file.
- Install additional RAM (available from your hardware supplier).

➨ Symbols not being looked up correctly

Fault:

- Symbol shadowing problem in the source code.
- The original symbol was destroyed because it was not referenced by code in the standalone application and a new symbol was created by **`read`** or **`intern`**.

Fixes

- Correct the source code.
- Check that the home package of the symbol is not the `toploop` package. This package is always deleted and its symbols have their values, function definitions and property lists removed.
- Preserve the package of the troublesome symbol and keep its internal or external symbols as appropriate using the `:package-directives` argument to **`create-standalone-application`**.
- One way to reference the troublesome symbol from your code is by putting it on the property list of the symbol naming your program entry point.

➨ Symbol unexpectedly not bound/fbound

This problem has the same possible causes and the same suggested fixes as **Symbols not being looked up correctly** above.

➨ `get` unexpectedly failing

This problem has the same possible causes and the same suggested fixes as **Symbols not being looked up correctly** above.

➨ Unexpected exit from the application

Fault:

- This problem usually occurs when the Lisp error handlers are removed and an error occurs, causing **`unwind-stack`** to unwind all the way out of the application.

Fixes

- Determine the cause of the error by keeping the error handlers, **`format`** and the printer. Correct the error or trap it and handle it yourself using **`trap-exits`**, **`errorset`** or `*error-hook*`.
- If the application will not run at all, remove fewer parts of the runtime system and try again.

➨ Apparent failure to run at all

The causes and suggested fixes are the same as **Unexpected exit from the application** above.

4.4.2 Specific Problems

➨ Image too large to load

A dialog box reporting **Image too large to load** appears.

Fault

- Insufficient memory available to read in the standalone application

Fixes

- Try to remove more from the standalone application.
- Install additional RAM (available from your hardware supplier).

➨ Stack overflow 1

A dialog box reporting the error **HARD ERROR: stack overflow** appears.

Fault

- Source code error (so it also happens if the code is loaded and executed in the full runtime system image).

- Lisp attempting to raise an error but the printer and/or **`format`** have been removed or the error handlers have been incorrectly redefined leading to more errors which cannot be handled.

Fixes

- Check for a source code error by loading the code into the full runtime system image and running it.
- Leave the printer, **`format`** and the error handlers intact and repeat the actions which led to the stack overflow. This may cause an error message to appear. Fix the cause of the error.
- Remove less of the system until the problem goes away. Thereafter, do not remove the parts of the system which prevent the problem recurring.

➨ Stack overflow 2

A dialog box reporting the error **Error: Stack overflow. You may be recursing infinitely.** appears. This problem has the same possible causes and the same suggested fixes as **Stack overflow 1** above.

➨ Trying to write to a closed stream

A dialog box reporting the error **Error: Argument #<CLOSED-STREAM ...> to i/o function must be to an open stream** appears.

Fault

- Trying to read/write to a stream which has been closed or to an object which is not a stream at all (a source code error).
- Lisp trying to write to `*terminal-io*`, `*error-output*` or another system stream. For instance, it may be attempting to print the text of an error message or warning. `*terminal-io*` corresponds to the Toploop window in a normal image and is not opened automatically in a standalone application.

Fixes

- Correct the source code error.
- Either open `*terminal-io*` and `*error-output*` as windows or sink all output sent to them by binding them to the result of `(make-broadcast-stream)`. See the entry on `*terminal-io*` in the Online Manual for more information.

➨ System error

An dialog saying **Sorry, a system error occurred** appears.

Fault

- The machine has got into an abnormal state through earlier errors.
- The user program has performed an illegal operation with Lisp's error checking turned off.
- The user program has performed an ill-conditioned operation using the Foreign Function Interface.
- The user program tried to perform an operation on an object of the wrong type (for example, taking the **`car`** or **`cdr`** of something mistakenly declared as a cons) while the safety level of the compiler was set inappropriately. (For information on controlling the compiler's safety level, read the file Controlling the Compiler in the folder Programming Examples supplied with the Allegro CL for Windows standard system.)

Fixes

- Reboot the machine and try again.
- Switch on the maximum compiler safety level. Check and correct the source code.

➨ System low-level debugger entered

The possible causes and suggested fixes are the same as for **System error** above.

[This page intentionally left blank.]

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

[This page intentionally left blank.]

Read This First

This document introduces Allegro CL 3.0 for Windows.

Allegro CL 3.0 for Windows runs on either Microsoft Windows 3.1 or Windows for Workgroups 3.11 (both of which require Win32s 1.30 or later), or on Windows NT Workstation 3.51 or Windows 95. Win32s 1.30 is included with the Allegro CL distribution. Note that the product is not supported on earlier versions of Win32s or versions of Windows NT prior to 3.51 Certain features are known not to work unless you have the specified or later version of Windows software. Please note that you must have a specially licensed version of Allegro CL 3.0 for Windows to run in Japan.

1 Inventory

Thank you for ordering Allegro CL 3.0 for Windows. In the box, along with this document you should have received:

- A CD jewel case. The license agreement is printed on this envelope. **Note that by opening the case, you are agreeing to the license agreement.** Please read the license agreement carefully.
- 2 volumes of wire-bound documentation.
- A Registration Card. Please send in this card! See section 2 **Registering yourself** below.
- The License Agreement. By opening the sealed CD jewel case, you are agreeing to the terms of the License Agreement. Please read it carefully.
- Win-Emacs order form. Win-Emacs is a Windows version of GNU Emacs. An interface to Allegro CL 3.0 for Windows is available.
- Professional upgrade order form. The Professional version of Allegro CL 3.0 for Windows has additional features and extended support.

Additional fliers describing other Franz Inc. products and services may also be included. This document and the items listed above are all that is required to install and use Allegro CL 3.0 for Windows.

2 Registering yourself

Please register by filling out the self-addressed registration form included in the Allegro CL 3.0 for Windows package and mailing it to Franz Inc.

Note that only registered users can take advantage of Franz customer support (see section **5 Support** below). Registered users also receive update notices and the *Franz Tempo* newsletter.

Copyright © 1995 by Franz Inc. All rights reserved. This is revision 1 of this document. This document has Franz Inc. part number D-F-00-PC0-01-51025-31s and is dated October 25, 1995. The following notices apply:
Allegro CL 3.0 for Windows is a trademark and Allegro CL is a registered trademark of Franz Inc. Windows, Windows 95, Windows for Workgroups, and MS Windows are trademarks of Microsoft.

3 Installation

Installation of Allegro CL 3.0 for Windows is described in section 2 **Installation** on page 2 of the *Getting Started* manual in volume 1 of the documentation.

Notes on installation:

- The Allegro CL distribution is on a CD. Win32s is also on the CD. Please contact Franz Inc. (ask for PC Sales) if you do not have a CD driver and need diskettes instead. See section 7 **Contacting Allegro CL Customer Support** for information on contacting Franz Inc.

4 Documentation

The Online Manual describes Common Lisp, Common Graphics and CLOS. There are two volumes of printed documentation.

Each printed volume contains more than one manual. The manuals are separated by tabs. Each manual has its own index and volume 2 contains a general index to all manuals. The first page of each volume describes the contents of both volumes.

Start with the *Getting Started* manual in volume 1. It tells you how to install Allegro CL 3.0 for Windows and gives other pertinent information. Also very useful when starting out is the *Programming Tools* manual in volume 2.

See section 3 **Documentation** of the *Getting Started* manual for more information on documentation for Allegro CL 3.0 for Windows.

See the Version 3.0 Release Notes entry in the Online Manual for information on what is new and/or changed. Display this entry by choosing **Manual Contents** from the Help menu after installing and starting Allegro CL 3.0 for Windows and click on the link to **Version 3.0 Release Notes**.

5 Support

There are two levels of support available for Allegro CL 3.0 for Windows. The first, sixty-day warranty support, is available to all purchasers of Allegro CL 3.0 for Windows. The second, Franz Premier Support, is available for an additional charge.

Sixty-day warranty support

All registered purchasers of Allegro CL 3.0 for Windows receive 60 days of free support, starting on the date of purchase. (See section 2 above for information on registering.) If you have problems during the first 60 days, you may contact Franz Inc. for assistance. See section 6 **Contacting Allegro CL Customer Support** below for more information on contacting Franz Inc.

Have the registration number (on the CD jewel case) handy when you call for warranty support.

Franz Premier Support

For an additional fee, Franz Premier Support is available for users of Allegro CL 3.0 for Windows. This program provides user with direct telephone and electronic mail support from Franz technical staff. Please contact Franz Inc. for more information regarding Franz Premier Support. Call + 510-548-3600 (in the USA) for information on Franz Premier Support. Ask for PC Sales.

World Wide Web site

Franz Inc. maintains a World Wide Web page. Its URL is

```
http://www.franz.com
```

The page contains information about Allegro CL 3.0 for Windows and other Franz Inc. products. Of particular interest to users of Allegro CL 3.0 for Windows is the ability to access the *Allegro CL 3.0 for Windows FAQ* and patches.

The FAQ (Frequently Asked Questions) is a document written in question and answer format. It is updated regularly, often with answers to questions that the Franz Inc. support staff notices are common. Hints and tips (about optimizing code, for example) are also provided. We recommend that users visit the WWW page from time to time and examine or download the current version of the FAQ. Patches are fixes to bugs and misfeatures in Allegro CL 3.0 for Windows. They are provided when necessary. They are described on and can be downloaded from the WWW page.

6 Things to note about Allegro CL 3.0 for Windows

In this section we list known problems with release 3.0. More information about bugs may also be available on Franz Inc. WWW page (see section 5 above).

1. **You must use Win32s 1.30 for certain features to work.** Common Control widgets and certain Allegro CL 3.0 for Windows dialogs require the functionality available in Win32s version 1.30 or later (1.30 is included with the distribution). If you are running Windows 3.1 or Windows for Workgroups, please be sure to install the version of Win32s supplied with the distribution. Windows 95 users do not need Win32s. Windows NT users do not need Win32s either, but must have version 3.51 or later.
2. **Compiled files must have extension** *fsl*. They will fail to load if they have a different extension.
3. **Compiled Lisp (*fsl*) files from earlier versions will not load**. *fsl* files generated by version 2.0 (or 1.0) cannot be loaded into Allegro CL 3.0 for Windows.

Windows 95/Windows NT do not support 16-bit DLL's

Only 32-bit DLL's can be linked to Allegro CL 3.0 for Windows when run under Windows 95 or Windows NT Workstation 3.51. (16-bit DLL's can be used when running under Windows 3.1 or Windows for Workgroups.)

Editor

The in-Lisp editor will only work with files smaller than 32K bytes.

Franz has teamed up with Pearl Software to supply Win-Emacs, a Windows port of GNU Emacs version 19. Please contact Franz for more information.

Note that Franz Inc. does not support Win-Emacs, except for problems arising from the interface between Allegro CL and Win-Emacs. Other problems with Win-Emacs should be referred to Pearl Software at the address in the Win-Emacs package. Pearl Software charges an annual fee for support.

The Toploop window (where the Lisp prompt first appears) is bound by the 32K limitation, but it will usually adjust itself (by deleting from the top) when the limit is reached. Cutting and pasting to that window may confuse the automatic adjustment, leading to an error. Calling for a new prompt (the leftmost item on the toolbar) will fix the problem.

Certain other dialogs (such as the History dialog and the Trace dialog) are not bound by the 32K limit. Long traces are better done in the new Trace dialog.

2.0 patches (however obtained) should not be used

Patches that version 2.0 customers may have obtained from Franz Inc. whether in compiled form (as *fsl* files) or in source form (perhaps from the FAQ) should not be loaded into version 3.0. In most cases, the functionality implemented by the patch is already in version 3.0. If not, please contact Franz Inc. for assistance.

Structure editor

The structure editor is no longer built into Allegro CL 3.0 for Windows. The directory *ex\structed* contains the source for the structure editor and a *readme.txt* file which explains how to load it (assuming you want its features).

Professional and standard versions incompatible

You have the standard version. Image (*img*) and compiled Lisp (*fsl*) files from the standard version will not load into the Professional version and image and compiled Lisp files from the Professional version will not load into the standard version.

7 Contacting Allegro CL customer support

All purchasers of Allegro CL 3.0 for Windows receive 60 days of support. Those customers who purchase Franz Premier Support (see section 5 above) are also eligible for support after the sixty-day warranty has expired.

If you have difficulties installing or using Allegro CL 3.0 for Windows, please do the following before you call customer support:

- Read the section in the documentation that contains information about the functions, macros, or special forms you are using.
- Look at the Allegro CL 3.0 for Windows FAQ (see section 5) to see if the problem is discussed there.

The best way to get support is to send e-mail to **pc-support@franz.com**. Please include your registration number.

You also may call Franz Inc. for support if you are still covered by the sixty-day warranty or have purchased Franz Premier Support. In either case, before you call, please:

- Be at your computer.
- Have your registration number handy (it is on the CD jewel case and the License Agreement).
- Have the exact text of any error messages that Allegro CL 3.0 for Windows displayed on your screen.
- Be able to describe your question or problem in detail.

The hours and telephone number of customer support are as follows. (The hours are subject to change without notice.)

Hours: 9 AM - 5 PM Pacific time, Monday through Friday (excluding Franz Inc. holidays)

Phone: + 510-548-3600 (customers outside North America, add international dialing access number and the country code for the USA)

Fax: + 510-548-8253 (anytime)

Ask for PC Sales rather than PC Technical Support if you need replacement items or you need diskettes in place of the CD.

8 Custom training and consulting services

Franz Inc. offers full training and custom software services. We offer corporate or group courses on programming with Common Lisp, CLOS, and CLIM (the Common Lisp Interface Manager) on various platforms. Franz also offers custom software services, including aid in porting, optimizing, or extending your application. Please contact Franz Inc. at the number above for more information on these services. Ask for PC Sales.

Read This First

This document introduces Allegro CL 3.0 for Windows Professional.

Allegro CL 3.0 for Windows runs on either Microsoft Windows 3.1 or Windows for Workgroups 3.11 (both of which require Win32s 1.30 or later), or on Windows NT Workstation 3.51 or Windows 95. Win32s 1.30 is included with the Allegro CL 3.0 for Windows distribution. Note that the product is not supported on earlier versions of Win32s or versions of Windows NT prior to 3.51 Certain features are known not to work unless you have the specified or later version of Windows software. Please note that you must have a specially licensed version of Allegro CL 3.0 for Windows to run in Japan.

1 Inventory

Thank you for ordering Allegro CL 3.0 for Windows. In the box, along with this document you should have received:

- A CD jewel case. The jewel case is sealed. **By opening the case, you are agreeing to the terms of the License Agreement.** Please read the License Agreement carefully.
- 2 volumes of wire-bound documentation.
- The *Professional Supplement* manual (not wire bound).
- A Registration Card. Please send in this card! See section 2 **Registering yourself and ordering source code** below.
- The License Agreement. By opening the sealed CD jewel case, you are agreeing to the terms of the License Agreement. Please read it carefully. The product serial number appears on the License Agreement.
- Source Code and Support Addendum. Sign and return this License Agreement addendum in order to receive certain Allegro CL 3.0 for Windows source code and to be eligible for full support. See section 2 **Registering yourself and ordering source code** below.
- Win-Emacs (in a separate box). Win-Emacs is a Windows version of GNU Emacs. The version supplied with Allegro CL 3.0 for Windows Professional Version includes the *editi* interface between Allegro CL 3.0 for Windows and Emacs. More information is contained in the Win-Emacs box.

Additional fliers describing other Franz Inc. products and services may also be included. This document and the items listed above are all that is required to install and use Allegro CL 3.0 for Windows.

2 Registering yourself and ordering source code

Please register by filling out the self-addressed registration form included in the Allegro CL 3.0 for Windows package and mailing it to Franz Inc. and by filling out the document entitled SOURCE CODE AND SUPPORT ADDENDUM TO FRANZ INC. ALLEGRO CL 3.0 FOR WINDOWS PROFESSIONAL SOFTWARE LICENSE AGREEMENT. **Only registered users can take advantage of Franz customer support**. The Addendum should be mailed to Franz Inc., 1995 University Ave., Berkeley, CA 94704, USA. It can also be faxed to + 510-548-8253. Be sure all items are

Copyright © 1995 by Franz Inc. All rights reserved. This is revision 1 of this document. This document has Franz Inc. part number D-F-00-PC0-01-51025-31p and is dated October 25, 1995. The following notices apply:
Allegro CL 3.0 for Windows is a trademark and Allegro CL is a registered trademark of Franz Inc. Windows, Windows 95, Windows for Workgroups, and MS Windows are trademarks of Microsoft.

filled in, including the serial number, the licensee information, and the shipping address at the end of the form. After the License Addendum is received, support will be activated and a diskette containing the relevant source code will be mailed to the shipping address specified on the License Addendum. See section 2.2 **Source code** in the *Professional Supplement* for more information.

> **Even if you received version 3.0 as a free upgrade, you must sign and return the License Addendum to receive support and the updated and additional source code modules that come with version 3.0.**

3 Installation

Installation of Allegro CL 3.0 for Windows is described in section 2 **Installation** on page 2 of the *Getting Started* manual in volume 1 of the documentation.

Notes on installation:

- The Allegro CL 3.0 for Windows distribution is on a CD. Win32s is also on the CD. Please contact Franz Inc. (ask for PC sales) if you do not have a CD driver and need diskettes instead. See section 7 **Contacting Allegro CL Customer Support** for information on contacting Franz Inc.

4 Documentation

The Online Manual describes Common Lisp, Common Graphics and CLOS. There are three volumes of printed documentation, two larger and wire bound, one (the *Professional Supplement*) smaller and not wire bound.

Each wire-bound volume contains more than one manual. The manuals are separated by tabs. Each manual has its own index and volume 2 contains a general index to all manuals in both wire-bound volumes (but not the *Professional Supplement*). The first page of each volume describes the contents of both volumes.

Start with the *Getting Started* manual in volume 1. It tells you how to install Allegro CL 3.0 for Windows and gives other pertinent information. Also very useful when starting out is the *Programming Tools* manual in volume 2.

See section 3 **Documentation** of the *Getting Started* manual for more information on documentation for Allegro CL 3.0 for Windows.

See the Version 3.0 Release Notes entry in the Online Manual for information on what is new and/or changed. Display this entry by choosing **Manual Contents** from the Help menu after installing and starting Allegro CL 3.0 for WIndows and click on the link to **Version 3.0 Release Notes**.

5 Support

Professional version customers are entitled to one year of product support (from the date of purchase). You must send the Registration Card and the License Addendum to be eligible for this support (see section 2 **Registering yourself and ordering source code**). See section 2.1 **Support** in the Professional Supplement manual for more information on support.

World Wide Web site

Franz Inc. maintains a World Wide Web page. Its URL is

```
http://www.franz.com
```

The page contains information about Allegro CL 3.0 for Windows and other Franz Inc. products. Of particular interest to users of Allegro CL 3.0 for Windows is the ability to access the *Allegro CL 3.0 for Windows FAQ* and patches.

The FAQ (Frequently Asked Questions) is a document written in question and answer format. It is updated regularly, often with answers to questions that the Franz Inc. support staff notices are common. Hints and tips (about optimizing code, for example) are also provided. We recommend that users visit the WWW page from time to time and

examine or download the current version of the FAQ. Patches are fixes to bugs and misfeatures in Allegro CL 3.0 for Windows. They are provided when necessary. They are described on and can be downloaded from the WWW page.

6 Things to note about Allegro CL 3.0 for Windows

In this section we list known problems with release 3.0. More information about bugs may also be available on Franz Inc. WWW page (see section 5 above).

1. **You must use Win32s 1.30 for certain features to work.** Common Control widgets and certain Allegro CL 3.0 for Windows dialogs require the functionality available in Win32s version 1.30 or later (1.30 is included with the distribution). If you are running Windows 3.1 or Windows for Workgroups, please be sure to install the version of Win32s supplied with the distribution. Windows 95 users do not need Win32s. Windows NT users do not need Win32s either, but must have version 3.51 or later.
2. **Compiled files must have extension** *fsl*. They will fail to load if they have a different extension.
3. **Compiled Lisp (*fsl*) files from earlier versions will not load**. *fsl* files generated by version 2.0 (or 1.0) cannot be loaded into Allegro CL 3.0 for Windows.

Windows 95/Windows NT do not support 16-bit DLL's

Only 32-bit DLL's can be linked to Allegro CL 3.0 for Windows when run under Windows 95 or Windows NT Workstation 3.51. (16-bit DLL's can be used when running under Windows 3.1 or Windows for Workgroups.)

Editor

The in-Lisp editor will only work with files smaller than 32K bytes.

Franz has teamed up with Pearl Software to supply Win-Emacs, a Windows port of GNU Emacs version 19. (You should have received a copy in a separate box.)

Note that Franz Inc. does not support Win-Emacs, except for problems arising from the interface between Allegro CL 3.0 for Windows and Win-Emacs. Other problems with Win-Emacs should be referred to Pearl Software at the address in the Win-Emacs package. Pearl Software charges an annual fee for support.

The Toploop window (where the Lisp prompt first appears) is bound by the 32K limitation, but it will usually adjust itself (by deleting from the top) when the limit is reached. Cutting and pasting to that window may confuse the automatic adjustment, leading to an error. Calling for a new prompt (the leftmost item on the toolbar) will fix the problem.

Certain other dialogs (such as the History dialog and the Trace dialog) are not bound by the 32K limit. Long traces are better done in the new Trace dialog.

2.0 patches (however obtained) should not be used

Patches that version 2.0 customers may have obtained from Franz Inc. whether in compiled form (as *fsl* files) or in source form (perhaps from the FAQ) should not be loaded into version 3.0. In most cases, the functionality implemented by the patch is already in version 3.0. If not, please contact Franz Inc. for assistance.

Structure editor

The structure editor is no longer built into Allegro CL 3.0 for Windows. The directory *ex\structed* contains the source for the structure editor and a *readme.txt* file which explains how to load it (assuming you want its features).

Professional and standard versions incompatible

You have the Professional version. Image (*img*) and compiled Lisp (*fsl*) files from the standard version will not load into the Professional version and image and compiled Lisp files from the Professional version will not load into the standard version.

7 Contacting Allegro CL customer support

If you have difficulties installing or using Allegro CL 3.0 for Windows, please do the following before you call customer support:

- Read the section in the documentation that contains information about the functions, macros, or special forms you are using.
- Look at the Allegro CL 3.0 for Windows FAQ to see if the problem is discussed there.

The best way to get support is to send e-mail to **pc-support@franz.com**. Please include your registration number.

You may also send a fax or call Franz Inc. for support. Before you call, please:

- Be at your computer.
- Have your registration number handy (it is on the CD jewel case and the License Agreement).
- Have the exact text of any error messages that Allegro CL 3.0 for Windows displayed on your screen.
- Be able to describe your question or problem in detail.

The hours and telephone number of customer support are as follows. (The hours are subject to change without notice.)

Hours: 9 AM - 5 PM Pacific time, Monday through Friday (excluding holidays)

Phone: + 510-548-3600 (customers outside North America, add international dialing access number and the country code for the USA)

Fax: + 510-548-8253 (anytime)

More information on support can be found ins section 2.1 **Support** of the *Professional Supplement* manual.

Ask for PC Sales rather than PC Technical Support if you need replacement items or you need diskettes in place of the CD.

8 Custom training and consulting services

Franz Inc. offers full training and custom software services. We offer corporate or group courses on programming with Common Lisp, CLOS, and CLIM (the Common Lisp Interface Manager) on various platforms. Franz also offers custom software services, including aid in porting, optimizing, or extending your application. Please contact Franz Inc. at the number above for more information on these services. Ask for PC Sales.